Fully polynomial-time parameterized computations for graphs and matrices of low treewidth[*]

Fedor V. Fomin[†] Daniel Lokshtanov[‡] Michał Pilipczuk[§] Saket Saurabh[¶]
Marcin Wrochna[‖]

Abstract

We investigate the complexity of several fundamental polynomial-time solvable problems on graphs and on matrices, when the given instance has low treewidth; in the case of matrices, we consider the treewidth of the graph formed by non-zero entries. In each of the considered cases, the best known algorithms working on general graphs run in polynomial time, however the exponent of the polynomial is large. Therefore, our main goal is to construct algorithms with running time of the form $\text{poly}(k) \cdot n$ or $\text{poly}(k) \cdot n \log n$, where k is the width of the tree decomposition given on the input. Such procedures would outperform the best known algorithms for the considered problems already for moderate values of the treewidth, like $\mathcal{O}(n^{1/c})$ for some small constant c.

Our results include:

– an algorithm for computing the determinant and the rank of an $n \times n$ matrix using $\mathcal{O}(k^3 \cdot n)$ time and arithmetic operations;

– an algorithm for solving a system of linear equations using $\mathcal{O}(k^3 \cdot n)$ time and arithmetic operations;

– an $\mathcal{O}(k^3 \cdot n \log n)$-time randomized algorithm for finding the cardinality of a maximum matching in a graph;

– an $\mathcal{O}(k^4 \cdot n \log^2 n)$-time randomized algorithm for constructing a maximum matching in a graph;

– an $\mathcal{O}(k^2 \cdot n \log n)$-time algorithm for finding a maximum vertex flow in a directed graph.

Moreover, we give an approximation algorithm for treewidth with time complexity suited to the running times as above. Namely, the algorithm, when given a graph G and integer k, runs in time $\mathcal{O}(k^7 \cdot n \log n)$ and either correctly reports that the treewidth of G is larger than k, or constructs a tree decomposition of G of width $\mathcal{O}(k^2)$.

The above results stand in contrast with the recent work of Abboud et al. [SODA 2016], which shows that the existence of algorithms with similar running times is unlikely for the problems of finding the diameter and the radius of a graph of low treewidth.

[*]D. Lokshtanov is supported by the BeHard grant under the recruitment program of the Bergen Research Foundation. The research of F. V. Fomin leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement n. 267959. S. Saurabh is supported by PARAPPROX, ERC starting grant no. 306992. The research of Mi. Pilipczuk and M. Wrochna is supported by Polish National Science Centre grant UMO-2013/11/D/ST6/03073. During the work on these results, Michał Pilipczuk held a post-doc position at Warsaw Center of Mathematics and Computer Science, and was supported by the Foundation for Polish Science (FNP) via the START stipend program.

[†]Department of Informatics, University of Bergen, Norway, fomin@ii.uib.no.

[‡]Department of Informatics, University of Bergen, Norway, daniello@ii.uib.no.

[§]Institute of Informatics, University of Warsaw, Poland, michal.pilipczuk@mimuw.edu.pl.

[¶]Institute of Mathematical Sciences, India, saket@imsc.res.in, and Department of Informatics, University of Bergen, Norway, Saket.Saurabh@ii.uib.no.

[‖]Institute of Informatics, University of Warsaw, Poland, m.wrochna@mimuw.edu.pl.

1 Introduction

The idea of exploiting small separators in graphs dates back to the early days of algorithm design, and in particular to the work of Lipton and Tarjan [44]. Namely, the Lipton-Tarjan planar separator theorem states that every n-vertex planar graph admits a separator of size $\mathcal{O}(\sqrt{n})$ that splits it in a balanced way, and which moreover can be found efficiently. Applying Divide & Conquer on small balanced separators is now a basic technique, commonly used when working with algorithms on "well-decomposable" graphs, including polynomial-time, approximation, and parameterized paradigms.

In structural graph theory, the concept of graphs that are well-decomposable using small separators, is captured by the notion of *treewidth*. Informally, the treewidth of a graph is the optimum width of its *tree decomposition*, which expresses the idea of breaking it into small pieces using small separators. Because discovering a low-width tree decomposition of a graph provides a uniform way to exploit the properties of small separators algorithmically, usually by means of dynamic programming or Divide & Conquer, treewidth became one of the fundamental concepts in graph algorithms. We refer to chapters in textbooks [21, Chapter 7], [63, Chapter 10], and [41, Chapter 10] for an introduction to treewidth and its algorithmic applications.

Treewidth is also important from the point of view of applications, as graphs of low treewidth do appear in practice. For instance, the control-flow graphs of programs in popular programming languages have constant treewidth [35, 60]. On the other hand, topologically-constrained graphs, like planar graphs or H-minor free graphs, have treewidth $\mathcal{O}(\sqrt{n})$.

Arguably, the usefulness of treewidth as a robust abstraction for the concept of being well-decomposable has been very well understood in the design of algorithm for NP-hard problems, and in particular in parameterized complexity. However, it seems that the applicability of treewidth for solving fundamental polynomial-time solvable problems, like Diameter, Maximum Matching or Maximum Flow, is relatively unexplored. Prior to our work, the research on polynomial-time algorithms on well-decomposable graphs was mainly focused on using Lipton-Tarjan-style separator theorems for specific graph classes, rather than treewidth and tree decompositions; see e.g. [3, 17, 49, 67]. We see two main reasons for this phenomenon.

First, the standard dynamic programming approach on graphs of low treewidth inherently requires time which is *exponential* in the treewidth of the graph. While the exponential dependence on the treewidth is unavoidable for NP-hard problems (unless P=NP), for polynomial-time solvable problems it would be much more desirable to have algorithms on a graph of treewidth k with running times of the form $f(k) \cdot n^c$, for some very small constant c and a *polynomial* function f. Consider for example the Maximum Matching problem. There is a wide variety of algorithms for finding a maximum matching in a graph; however, their running times are far from linear. On the other hand, it is not hard to obtain an algorithm for Maximum Matching on a graph, given together with a tree decomposition of width k, that runs in time $\mathcal{O}(3^k \cdot k^{\mathcal{O}(1)} \cdot n)$; this outperforms all the general-purpose algorithms for constant values of the treewidth. But is it possible to obtain an algorithm with running time $\mathcal{O}(k^d \cdot n)$ or $\mathcal{O}(k^d \cdot n \log n)$ for some constant d, implying a significant speed-up already for moderate values of treewidth, like $k = \mathcal{O}(n^{1/3d})$? This question can be asked also for a number of other problems for which the known general-purpose polynomial-time algorithms have unsatisfactory running times. As mentioned earlier, standard dynamic programming on tree decompositions seems difficult to apply here, due to inherently exponential number of states. Interestingly, the recent work of Abboud et al. [1] indicates that for some polynomial-time solvable problems this seems to be a real obstacle. In particular, Abboud et al. [1] proved that the Diameter and Radius problems can be solved in time $2^{\mathcal{O}(k \log k)} \cdot n^{1+o(1)}$ on graphs of treewidth k, but achieving running time of the form $2^{o(k)} \cdot n^{2-\varepsilon}$ for any $\varepsilon > 0$ for Diameter would already contradict the Strong Exponential Time Hypothesis (SETH) of Impagliazzo et al. [38]; the same lower bound is also given for Radius, but under a stronger assumption.

Second, in order to use the treewidth effectively, we need efficient algorithms to construct low width decompositions. Computing treewidth exactly is NP-hard [5]. Standard parameterized algorithms for approximating treewidth [8, 10, 57] have exponential running time dependency on the target width, while known polynomial-time approximation algorithms, e.g. [27], are based on heavy tools like semi-definite programming, and hence their running time is far from linear with respect to the graph size. Coming back to our example with Maximum Matching, to find a tree decomposition of width at most k, we either have to run an algorithm with running time exponential in k or to use a polynomial time treewidth approximation whose running time is much worse than the time of any reasonable algorithm solving Maximum Matching directly.

Thus, in order to understand the applicability of treewidth as a tool for polynomial-time solvable problems, we have to develop a new algorithmic toolbox and new approximation algorithms suitable for such purposes. The goal of this paper is to provide basic answers to the questions above, and thus to initiate a systematic study of the treewidth parameterization for fundamental problems that are solvable in polynomial time, but for which the fastest known general-purpose algorithms have unsatisfactory running times. Examples of such problems include Maximum Matching, Maximum Flow, and various algebraic problems on matrices, like computing determinants or solving systems of linear equations. For such problems, our main concrete goal is to design an algorithm with running time of the form $\mathcal{O}(k^d \cdot p(n))$ for some constant d and polynomial $p(n)$ that would be much smaller than the running time bound of the fastest known unparameterized algorithm. Mirroring the terminology of parameterized complexity, we will call such algorithms *fully polynomial FPT* (FPT stands for *fixed-parameter tractable*). Although several results of this kind are scattered throughout the literature [2, 14, 15, 16, 53], mostly concerning shortest path problems, no systematic investigations have been made so far.

Our contribution. Our first main result is a new approximation algorithm for treewidth, which is suited to the type of running times at which we aim.

Theorem 1.1. *There exists an algorithm that, given a graph G on n vertices and a positive integer k, in time $\mathcal{O}(k^7 \cdot n \log n)$ either provides a tree decomposition of G of width at most $\mathcal{O}(k^2)$, or correctly concludes that $\mathtt{tw}(G) \geq k$.*

Thus, Theorem 1.1 can serve the same role for fully polynomial FPT algorithms parameterized by treewidth, as Bodlaender's linear-time algorithm for treewidth [8] serves for Courcelle's theorem [20]: it can be used to remove the assumption that a suitable tree decomposition is given on the input, because such a decomposition can be approximated roughly within the same running time.

Next, we turn to algebraic problems on matrices. Given an $n \times m$ matrix A over some field, we can construct a bipartite graph G_A as follows: the vertices on the opposite sides of the bipartition correspond to rows and columns of A, respectively, and a row is adjacent to a column if and only if the entry on their intersection is non-zero in A. Then, we can investigate the complexity of computational problems when a tree decomposition of G_A of (small) width k is given on the input. As a graph on n vertices and of treewidth k has at most kn edges, it follows that such matrices are sparse — they contain only $\mathcal{O}(kn)$ non-zero entries. It is perhaps more convenient to think of them as edge-weighted bipartite graphs: we label each edge of G_A with the element placed in the corresponding entry of the matrix. We assume the matrix is given in sparse form, e.g., as adjacency lists for G_A.

Our main result here is a pivoting scheme that essentially enables us to perform Gaussian elimination on matrices of small treewidth. In particular, we are able to obtain useful factorizations of such matrices, which gives us information about the determinant and rank, and the possibility to solve linear equations efficiently. We cannot expect to invert matrices in near-linear time, as even very simple matrices have inverses with $\Omega(n^2)$ entries (e.g. the square matrix with $M[j,i]=1$ for $i-j\in\{0,1\}$, 0 elsewhere). The following theorems gather the main corollaries of our results; we refer to Sections 2 and 4 for definitions of pathwidth, tree-partition width, and more details on the form of factorizations that we obtain. Note that for square matrices, the same results can be applied to decompositions of the usual symmetric graph, as explained in Section 2.

Theorem 1.2. *Given an $n\times m$ matrix M over a field $\mathbb{F}$ and a path or tree-partition decomposition of its bipartite graph G_M of width k, Gaussian elimination on M can be performed using $\mathcal{O}(k^2\cdot(n+m))$ field operations and time. In particular, the rank, determinant, a maximal nonsingular submatrix and a PLUQ-factorization can be computed in this time. Furthermore, for every $r\in\mathbb{F}^n$, the system of linear equations $Mx=r$ can be solved in $\mathcal{O}(k\cdot(n+m))$ additional field operations and time.*

Theorem 1.3. *Given an $n\times m$ matrix M over a field $\mathbb{F}$ and a tree-decomposition of its bipartite graph G_M of width k, we can calculate the rank, determinant and a generalized LU-factorization of M in $\mathcal{O}(k^3\cdot(n+m))$ field operations and time. Furthermore, for every $r\in\mathbb{F}^n$, the system of linear equations $Mx=r$ can be solved in $\mathcal{O}(k^2\cdot(n+m))$ additional field operations and time.*

Our algorithms work more efficiently for parameters pathwidth and tree-partition width, which can be larger than treewidth. The reason is the pivoting scheme underlying Theorems 1.2 and 1.3 works perfectly for path and tree-partition decompositions, whereas for standard tree decomposition the scheme can possibly create a lot of new non-zero entries in the matrix. However, we show how to reduce the case of tree decompositions to tree-partition decompositions by adjusting the idea of matrix sparsification for nested dissection of Alon and Yuster [3] to the setting of tree decompositions. Unfortunately, this reduction incurs an additional k factor in the running times of our algorithms, and we obtain a less robust factorization.

Observe that one can also use the known inequality $\mathtt{pw}(G)\leq\mathtt{tw}(G)\cdot\log_2 n$ (see e.g. [11]) to reduce the treewidth case to the pathwidth case. This trades the additional factor k for a factor $(\log n)^2$; depending on the actual value of k, this might be beneficial for the overall running time.

Note that Theorems 1.2 and 1.3 work over any field $\mathbb{F}$. Hence, we can use them to develop an algebraic algorithm for the maximum matching problem, using the classic approach via the Tutte matrix. This requires working in a field $\mathbb{F}$ (say, $\mathbb{F}_p$) of polynomial size, and hence the complexity of performing arithmetic operations in this field depends on the computation model. Below we count all such operations as constant time, and elaborate on this issue in Section 5.

Theorem 1.4. *There exists an algorithm that, given a graph G together with its tree decomposition of width at most k, uses $\mathcal{O}(k^3\cdot n)$ time and field operations and computes the size of a maximum matching in G. The algorithm is randomized with one-sided error: it is correct with probability at least $1-1/n^c$ for an arbitrarily chosen constant c, and in the case of an error it reports a suboptimal value.*

Theorem 1.4 only provides the size of a maximum matching; to construct the matching itself, we need some more work.

Theorem 1.5. *There exists an algorithm that, given a graph G together with its tree decomposition of width at most k, uses $\mathcal{O}(k^4\cdot n\log n)$ time and field operations and computes a maximum matching in G. The algorithm is randomized with one-sided error: it is correct with probability at least $1-1/n^c$ for an arbitrarily chosen constant c, and in the case of an error it reports a failure or a suboptimal matching.*

We remark that our algebraic approach is tailored to unweighted graphs, and cannot be easily extended to the weighted setting.

Finally, we turn our attention to the maximum flow problem. We prove that for vertex-disjoint flows we can also design a fully polynomial FPT algorithm with near-linear running time dependence on the size of the input. The algorithm works even on directed graphs (given a tree decomposition of the underlying undirected graph), but only in the unweighted setting (i.e., with unit vertex capacities, which boils down to finding vertex-disjoint paths).

Theorem 1.6. *There exists an algorithm that given an unweighted directed graph G on n vertices, distinct terminals $s, t \in V(G)$ with $(s,t) \notin E(G)$, and a tree decomposition of G of width at most k, works in time $\mathcal{O}(k^2 \cdot n \log n)$ and computes a maximum (s,t)-vertex flow together with a minimum (s,t)-vertex cut in G.*

Theorem 1.6 states the result only for single-source and single-sink flows, but it is easy to reduce other variants, like (S,T)-flows, to this setting. Note that in particular, Theorem 1.6 provides an algorithm for the maximum matching problem in bipartite graphs that is faster than the general one from Theorem 1.5: one just needs to add a new source s and a new sink t to the graph, and make s and t fully adjacent to the opposite sides of the bipartition.

Related work on polynomial-time algorithms on small treewidth graphs. The reachability and shortest paths problems on low treewidth graphs have received considerable attention in the literature, especially from the point of view of data structures [2, 15, 16, 14, 53]. In these works, the running time dependence on treewidth is either exponential or polynomial, which often leads to interesting trade-off questions. For most of these problems, classical algorithms already gave optimal time bounds up to logarithmic factors, but empirical studies suggest practical applicability of leveraging low treewidth even in that case [53]. However, the only instance we are aware of where an asymptotic improvement follows already for polynomially small treewidth is an $\mathcal{O}(k^2 \cdot n \log n)$ algorithm for computing the so called *almost-sure reachability set* in Markov decision processes by Chatterjee and Łącki [14], improving over a general $\mathcal{O}(m\sqrt{m}) = \mathcal{O}(k^{1.5} \cdot n^{1.5})$ algorithm.

As far as computation of maximum flows is concerned, we are aware only of the work of Hagerup et al. on multicommodity flows [36]; however, their approach inevitably leads to exponential running time dependence on the treewidth. The work of Hagerup et al. [36] was later used by Chambers and Eppstein [13] for the maximum flow problem in one-crossing-minor-free graphs; unfortunately, the exponential dependency on the size of the excluded minor persists.

Related work on approximating treewidth. Computing treewidth exactly is NP-hard [5], and moreover there is no constant-factor approximation for treewidth unless the Small Set Expansion Hypothesis fails [66]. However, when we allow the algorithm to run in FPT time when parameterized by the target width, then there is a wide variety of exact and approximation algorithms. Perhaps the best known are: the 4-approximation algorithm in $2^{\mathcal{O}(k)} \cdot n^2$ time of Robertson and Seymour [57] (see [21, 41] for an exposition of this algorithm) and the linear-time exact algorithm of Bodlaender with running time $k^{\mathcal{O}(k^3)} \cdot n$ [8]. Recently, Bodlaender et al. [10] obtained a 3-approximation in time $2^{\mathcal{O}(k)} \cdot n \log n$ and a 5-approximation in time $2^{\mathcal{O}(k)} \cdot n$. Essentially all the known approximation algorithms for treewidth follow the approach of Robertson and Seymour [57], which is based on recursively decomposing subgraphs by breaking them using balanced separators.

As far as polynomial-time approximation algorithms are concerned, the best known algorithm is due to Feige et al. [27] and it achieves approximation factor $\mathcal{O}(\sqrt{\log OPT})$ in polynomial time. Unfortunately, the running time is far from linear due to the use of semi-definite programming for the crucial subroutine of finding balanced separators; this is also the case in previous works [43, 4], which are based on linear programming as well.

For this reason, in the proof of Theorem 1.1 we develop a purely combinatorial $\mathcal{O}(OPT)$-factor approximation algorithm for finding balanced separators. This algorithm is based on the techniques of Feige and Mahdian [28], which are basic enough so that they can be implemented within the required running time. The new approximation algorithm for balanced separators is then combined with a trick of Reed [56]. Essentially, the original algorithm of Robertson and Seymour [57] only breaks the (small) interface between the subgraph being decomposed and the rest of the graph, which may result in $\Omega(n)$ recursion depth. Reed [56] observed that one can add an additional step of breaking the whole subgraph in a balanced way, which reduces the recursion depth to logarithmic and results in improving the running time dependence on the input size from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, at the cost of solving a more general (and usually more difficult) subproblem concerning balanced separators. Fortunately, our new approximation algorithm for balanced separators is flexible enough to solve this more general problem as well, so we arrive at $\mathcal{O}(n \log n)$ running time dependence on n.

Related work on matrix computations. Solving systems of linear equations and computing the determinant and rank of a matrix are ubiquitous, thoroughly explored topics in computer science, with a variety of well-known applications. Since sparse matrices often arise both in theory and in practice, the possibility (and often necessity) of exploiting their sparsity has been deeply studied as well. Here we consider matrices as sparse when their non-zero entries are not only few (that is, $o(n^2)$), but furthermore they are structured in a way that could potentially be exploited using graph-theoretical techniques. The two best known classical approaches in this direction are standard Gaussian elimination with a *perfect elimination ordering*, and *nested dissection*.

A common assumption in both approaches is that throughout the execution of an algorithm, no accidental cancellation occurs – that is, except for situations guaranteed and required by the algorithm, arithmetic operations never change a non-zero value to a zero. This can be assumed in some settings, such as when the input matrix is positive definite. Otherwise, as soon as accidental zeroes occur (for example, simply starting with a zero diagonal) some circumvention is required by finding a different pivot than originally planned. In practice, especially when working over real-valued matrices, one may expect this not to extend resource usage too much, but when working over finite fields it is clear that this assumption cannot be used to justify any resource bounds. Surprisingly, we are not aware of any work bounding the worst-case running time of an algorithm for the determinant of a matrix of small treewidth (or pathwidth) without this assumption. This may in part be explained by the fact that a better understanding of sparseness in graph theory and the rise of treewidth in popularity came after the classical work on sparse matrices, and by the reliance on heuristics in practice.

Perfect elimination ordering, generally speaking, refers to an ordering of rows and columns of a matrix such that Gaussian elimination introduces no *fill-in* – entries in the matrix where a zero entry becomes non-zero. A seminal result of Parter [51] and Rose [58] says that such an ordering exists if and only if the (symmetric) graph of the matrix is chordal (or *triangulated*, that is, every cycle with more than three edges has a chord). This assumes no accidental cancellation occurs. Hence to minimize space usage, one would search for a minimum completion to a chordal graph (the Minimum Fill-in problem), while to put a guarantee on the time spent on eliminating, one could demand a chordal completion with small cliques (a.k.a. small *frontsize*), which is equivalent to the graph of the original matrix having small treewidth [11]. Radhakrishnan et al. [55] use this approach to give an $\mathcal{O}(k^2 n)$ algorithm for solving systems of linear equations defined by matrices of treewidth k, assuming no accidental cancellation.

To lift this assumption one has to consider arbitrary pivoting and the bipartite graph of a matrix instead (with separate vertices for each row and each column), which also allows the study of non-symmetric, non-square matrices. A Γ-free ordering is an ordering of rows and columns of a matrix such that no $\left(\begin{smallmatrix} \star & \star \\ \star & 0 \end{smallmatrix}\right)$ submatrix occurs – it can be seen that such an ordering allows to perform Gaussian elimination with no fill-in ($\star$ represents a non-zero entry). This corresponds to a *strong ordering* of the bipartite graph G_M of the matrix – an ordering $\preccurlyeq$ of vertices such that for all vertices i, j, k, ℓ, if $i \preccurlyeq \ell$, $j \preccurlyeq k$, and $ji, ki, j\ell$ are edges, then $k\ell$ must be an edge too. A bipartite graph is known to be *chordal bipartite* graph (defined as a bipartite graph with no chordless cycles strictly longer than 4 – note it need not be chordal) if and only if it admits a strong ordering (see e.g. [23]). Golumbic and Goss [34] first related chordal bipartite graphs to Gaussian elimination with no fill-in, but assuming no accidental cancellation. Bakonyi and Bono [6] showed that when an accidental cancellation occurs and a pivot cannot be used, a different pivot can always be found. However, they do not consider the running time needed for finding the pivot, nor the number of arithmetic operations performed.

Nested dissection is a Divide & Conquer approach introduced by Lipton, Tarjan and Rose [44] to solve a system of linear equations whose matrix is symmetric positive definite and whose graph admits a certain separator structure. Intuitively, a *weak separator tree* for a graph gives a small separator of the graph that partitions its vertices into two balanced parts, which after removing the separator, are recursively partitioned in the same way. In work related to nested dissection, a *small* separator means one of size $\mathcal{O}(n^\gamma)$, where n is the number of remaining vertices of the graph and $\gamma < 1$ is a constant ($\gamma = \frac{1}{2}$ for planar and H-minor-free graphs). Thus an algorithm needs to handle a logarithmic number of separators whose total size is a geometric series bounded again by $\mathcal{O}(n^\gamma)$. In modern graph-theoretic language, this most closely corresponds to a (balanced, binary) tree-depth decomposition of depth $\mathcal{O}(n^\gamma)$.

To use nested dissection for matrices A that are not positive definite (so without assuming no accidental cancellation), Mucha and Sankowski [49] used it on AA^T instead, carefully recovering some properties of A afterwards. In order to guarantee a good separator structure for AA^T, however, they first need to decrease the degree of the graph of A by an approach called *vertex splitting*, introduced by Wilson [64]. Vertex splitting is the operation of replacing a vertex v with a path on three vertices v', w, v'' and replacing each incident edge uv with either uv' or uv''. It is easy to see that this operation preserves the number of perfect matchings, for example. The operation applied to the graph of a matrix can in fact be performed on the matrix, preserving its determinant too. By repeatedly splitting a vertex, we can transform it, together with incident edges, into a tree of degree bounded by 3. Choosing an appropriate partition of the incident edges, the structure of the graph can be preserved; for example, the knowledge of a planar embedding can be used to stay in the class of planar graphs. This allowed Mucha and Sankowski [49] to find maximum matchings via Gaussian elimination in planar graphs in $\mathcal{O}(n^{\omega/2})$ time, where $\omega < 2.38$ is the exponent of the best known matrix multiplication algorithm. Yuster and Zwick [67] showed that the weak separator tree structure can be preserved too, which allowed them to extend this result to H-minor-free graphs.

These methods were further extended by Alon and Yuster [3] to use vertex splitting and nested dissection on AA^T for solving arbitrary systems of linear equations over any field, for matrices whose graphs admit a weak separator tree structure. If the separators are of size $\mathcal{O}(n^\beta)$ and can be efficiently found, the algorithm works in $\mathcal{O}(n^{\omega\beta})$ time. However, it is randomized and very involved, in particular requiring arithmetic computations in field extensions of polynomial size. A careful translation of their proofs to tree decompositions could only give an $\mathcal{O}(k^5 \cdot n \log^3 n)$ randomized algorithm for matrices of treewidth k (that is, $\mathcal{O}(n'k'^2)$ [55] where $n' = nk$ and $k' = k^2 \log n$ after vertex splitting, with the recursion in [3] giving an additional $\log n$ factor).

Our approach differs in that for matrices with path or tree-partition decompositions of small width we show that a strong ordering respecting the decomposition can be easily found, and standard Gaussian elimination is enough, as long as the ordering is properly used when pivoting. For matrices with small treewidth this does not seem possible (an apparent obstacle here is that not all chordal graphs have strong orderings). However, a variant of the vertex splitting technique guided with a tree-decomposition allows us to simply (in particular, deterministically) reduce to the tree-partition case (instead of considering AA^T).

Related work on maximum matchings. The existence of a perfect matching in a graph can be tested by calculating the determinant of the Tutte matrix [61]. Lovász [45] showed that the size of a maximum matching can be found by computing the rank, while Mucha and Sankowski [48, 49] gave a randomized algorithm for extracting a maximum matching: in $\mathcal{O}(n^\omega)$ time for general graphs and $\mathcal{O}(n^{\omega/2})$ for planar graphs. The results on general graphs were later simplified by Harvey [37]. Before that, Edmonds [24] gave the first polynomial time algorithm, then bested by combinatorial algorithms of Micali and Vazirani [54, 62], Blum [7], and Gabow and Tarjan [31], each running in $\mathcal{O}(m\sqrt{n})$ time. Recently Mądry [46] gave an $\mathcal{O}(m^{10/7})$ algorithm for the unweighted bipartite case, then generalized to the weighted bipartite case by Cohen et al. [19]. For graphs of treewidth k, simply because their number of edges is $m = \mathcal{O}(kn)$, the above gives $\mathcal{O}(kn^{1.5})$ and $\mathcal{O}(k^{1.42}n^{1.42})$ algorithms, respectively.

Related work on maximum flows. The maximum flow problem is a classic subject with a long and rich literature. Starting with the first algorithm of Ford and Fulkerson [30], which works in time $\mathcal{O}(F \cdot (n+m))$ for integer capacities, where F is the maximum size of the flow, a long chain of improvements and generalizations was proposed throughout the years; see e.g. [19, 22, 25, 26, 33, 39, 46, 47, 50]. The running times of these algorithms vary depending on the variants they solve, but all of them are far larger than linear. In particular, the fastest known algorithm in the directed unit-weight setting, which is the case considered in this work, is due to Mądry [46, 47] and works in time $\mathcal{O}(m^{10/7})$. For this reason, recently there was a line of work on finding near-linear $(1+\varepsilon)$-approximation algorithms for the maximum flow problem [18, 40, 42, 52, 59], culminating in a $(1+\varepsilon)$-approximation algorithm working in undirected graphs in time $\mathcal{O}(\varepsilon^{-2} \cdot m \log^{11} n)$, proposed by Peng [52].

Outline. In Section 2 we establish notation and recall basic facts about matrices, flows, and tree-like decompositions of graphs. Section 3 is devoted to the approximation algorithm for treewidth, i.e., Theorem 1.1. In Section 4 we give our results for problems on matrices of low width, and in particular we prove Theorems 1.2 and 1.3. In Section 5 we apply these results to the maximum matching problem, proving Theorems 1.4 and 1.5. Section 6 is focused on the maximum vertex flow problem and contains a proof of Theorem 1.6. Finally, in Section 7 we gather short concluding remarks and state a number of open problems stemming from our work.

2 Preliminaries

Notation. We use standard graph notation; cf. [21]. All the graphs considered in this paper are simple, i.e., they have no loops or multiple edges connecting the same endpoints. For a graph G and $X \subseteq V(G)$, by $N_G[X]$ we denote the *closed neighborhood* of X, i.e., all the vertices that are either in X or are adjacent to vertices of X, and by $N_G(X) = N_G[X] \setminus X$ we denote the *open neighborhood* of X. When G is clear from the context, we drop the subscript. For a path P, the *internal vertices* of P are all the vertices of P apart from the endpoints. Paths P and Q are *internally vertex-disjoint* if no vertex of P is an internal vertex of Q and vice-versa. The set of connected components of a graph G is denoted by $\mathtt{cc}(G)$.

By $G[X]$ we denote the subgraph of G induced by X, and we define $G - X = G[V(G) \setminus X]$. Graph H is a *subgraph* of G, denoted $H \subseteq G$, if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. We say H is a *completion* of G if $V(H) = V(G)$ and $E(H) \supseteq E(G)$.

A 1-*subdivision* of a graph G is obtained from G by taking every edge $uv \in E(G)$, and replacing it with a new vertex w_{uv} and edges uw_{uv} and $w_{uv}v$.

Matrices. For an $n \times m$ matrix M, the entry at the intersection of the r-th row and c-th column is denoted as $M[r,c]$. For sets $X \subseteq \{1,\dots,n\}$ and $Y \subseteq \{1,\dots,m\}$, by $[M]_{X,Y}$ we denote the $|X| \times |Y|$ matrix formed by the entries of M appearing on the intersections of rows of X and columns of Y.

The *symmetric graph* of an $n \times n$ matrix (i.e., square, but not necessarily symmetric) is the undirected graph with vertices $\{1,\dots,n\}$ and an edge between i and j whenever $M[i,j] \neq 0$ or $M[j,i] \neq 0$. The *bipartite graph* of an $n \times m$ matrix is a bipartite, undirected graph with vertices in $\{r_1,\dots,r_n\} \cup \{c_1,\dots,c_m\}$ and an edge between r_i and c_j whenever $M[i,j] \neq 0$.

In this work, for describing the structure of a matrix M, we use the bipartite graph exclusively and denote it G_M, as it allows to express our results for arbitrary (not necessarily square) matrices. Note that any tree decomposition of the symmetric graph of a square matrix M can be turned into a decomposition of G_M of twice the width plus 1, by putting both the i-th row and i-th column in the same bag where index i was.

A matrix is in (non-reduced) *row-echelon form* if all zero rows (with only zero entries) are below all non-zero rows, and the leftmost non-zero coefficient of each row is strictly to the right of the leftmost non-zero coefficients of rows above it. In particular, there are no non-zero entries below the diagonal. We define column-echelon form analogously. A *PLUQ-factorization* of an $n \times m$ matrix M (also known as an LU-factorization with full pivoting) is a quadruple of matrices where: P is a permutation $n \times n$ matrix, L is an $n \times n$ matrix in column-echelon form with ones on the diagonal, U is an $n \times m$ matrix in row-echelon form, Q is a permutation $m \times m$ matrix, and $M = PLUQ$. A *generalized LU-factorization* of M is a sequence of matrices such that their product is M and each is either a permutation matrix or a matrix in row- or column-echelon form.

Flows and cuts. For a graph G and disjoint subsets of vertices $S, T \subseteq V(G)$, an *(S,T)-path* is a path in G that starts in a vertex of S, ends in a vertex of T, and whose internal vertices do not belong to $S \cup T$. In this paper, an *(S,T)-vertex flow* is a family of (S,T)-paths $\mathcal{F} = \{P_1, P_2, \dots, P_k\}$ that are internally vertex-disjoint; note that we do allow the paths to share endpoints in S or T. The size of a flow $\mathcal{F}$, denoted $|\mathcal{F}|$, is the number of paths in it. A subset $X \subseteq V(G) \setminus (S \cup T)$ is an *(S,T)-vertex cut* if no vertex of T is reachable by a path from some vertex of S in the graph $G - X$. A variant of the well-known Menger's theorem states that the maximum size of an (S,T)-vertex flow is always equal to the minimum size of an (S,T)-vertex cut, provided there is no edge between S and T. In case G is directed, instead of undirected paths, we consider directed paths starting from S and ending in T, and the same statement of Menger's theorem holds (the last condition translates to the nonexistence of edges from S to T). Note that in this definitions we are only interested in flows and cuts in unweighted graphs, or in other words, we put unit capacities on all the vertices.

There is a wide variety of algorithms for computing the maximum vertex flows and minimum vertex cuts in undirected/directed graphs in polynomial time. Among them, the most basic is the classic algorithm of Ford and Fulkerson, which uses the technique of finding consecutive augmentations of an (S,T)-vertex flow, up to the moment when a maximum flow is found. More precisely, the following well-known result is used.

Theorem 2.1 (Max-flow augmentation). *There exists an algorithm that, given a directed graph G on n vertices and m edges, disjoint subsets $S, T \subseteq V(G)$ with no edge from S to T, and some (S,T)-vertex flow $\mathcal{F}$, works in $\mathcal{O}(n+m)$ time and either certifies that $\mathcal{F}$ is maximum by providing an (S,T)-vertex cut of size $|\mathcal{F}|$, or finds an (S,T)-vertex flow $\mathcal{F}'$ with $|\mathcal{F}'| = |\mathcal{F}| + 1$.*

The classic proof of Theorem 2.1 works as follows: the algorithm first constructs the *residual network* that encodes where more flow could be pushed. Then, using a single BFS it looks for an *augmenting path*. If such an augmenting path can be found, then it can be used to modify the flow so that its size increases by one. On the other hand, the nonexistence of such a path uncovers an (S,T)-vertex cut of size $|\mathcal{F}|$. Of course, the analogue of Theorem 2.1 for undirected graphs follows by turning an undirected graph into a directed one by replacing every edge uv with arcs (u,v) and (v,u).

The next well-known corollary follows by applying the algorithm of Theorem 2.1 at most $k+1$ times.

Corollary 2.2. *There exists an algorithm that, given an undirected/directed graph G on n vertices and m edges, disjoint subsets $S,T \subseteq V(G)$ with no edge from S to T, and a positive integer k, works in time $\mathcal{O}(k \cdot (n+m))$ and provides one of the following outcomes:*

(a) a maximum (S,T)-vertex flow of size ℓ together with a minimum (S,T)-vertex cut size ℓ, for some $\ell \leq k$; or

(b) a correct conclusion that the size of the maximum (S,T)-vertex flow (equivalently, of the minimum (S,T)-vertex cut) is larger than k.

Tree decompositions. We now recall the main concepts of graph decompositions used in this paper. First, we recall standard tree and path decompositions.

Definition 2.3. A *tree decomposition* of a graph G is a pair $(\mathcal{T}, \{B_x\}_{x \in V(\mathcal{T})})$, where $\mathcal{T}$ is a tree and each node x of $\mathcal{T}$ is associated with a subset of vertices $B_x \subseteq V(G)$, called the *bag at x*. Moreover, the following condition have to be satisfied:

- For each edge $uv \in E(G)$, there is some $x \in V(\mathcal{T})$ such that $\{u,v\} \subseteq B_x$.
- For each vertex $u \in V(G)$, define $\mathcal{T}[u]$ to be the subgraph of $\mathcal{T}$ induced by nodes whose bags contain u. Then $\mathcal{T}[u]$ is a non-empty and connected subtree of $\mathcal{T}$.

The *width* of $\mathcal{T}$ is equal to $\max_{x \in V(\mathcal{T})} |B_x| - 1$, and the *treewidth of* G, denoted $\mathtt{tw}(G)$, is the minimum possible width of a tree decomposition of G. In case $\mathcal{T}$ is a path, we call $\mathcal{T}$ also a *path decomposition* of G. The *pathwidth of* G, denoted $\mathtt{pw}(G)$, is the minimum possible width of a tree decomposition of G.

We follow the convention that whenever $(\mathcal{T}, \{B_x\}_{x \in V(\mathcal{T})})$ is a tree decomposition of G, then elements of $V(\mathcal{T})$ are called *nodes* whereas elements of $V(G)$ are called *vertices*. Moreover, we often identify the nodes of $\mathcal{T}$ with bags associated with them, and hence we can talk about adjacent bags, etc. Also, we often refer to the tree $\mathcal{T}$ only as a tree decomposition, thus making the associated family $\{B_x\}_{x \in V(\mathcal{T})}$ of bags implicit.

Throughout this paper we assume that all tree or path decompositions of width k given on input have $\mathcal{O}(|V(G)|)$ nodes. In fact, we will assume that the input decompositions are *clean*, defined as follows.

Definition 2.4. A tree decomposition $(\mathcal{T}, \{B_x\}_{x \in V(\mathcal{T})})$ of G is called *clean* if for every $xy \in V(\mathcal{T})$, it holds that $B_x \nsubseteq B_y$ and $B_y \nsubseteq B_x$.

It is known that a clean tree decomposition of a graph on n vertices has at most n nodes, and that any tree (path) decomposition $\mathcal{T}$ of width k can be transformed in time $\mathcal{O}(k|V(\mathcal{T})|)$ to a *clean* tree (path) decomposition of the same width (see e.g. [29, Lemma 11.9]). Thus, the input decomposition can be always made clean in time linear in its size.

Finally, we recall the definition of another width parameter we use, see e.g. [65].

Definition 2.5. A *tree-partition decomposition* of a graph G is a pair $(\mathcal{T}, \{B_x\}_{x \in V(\mathcal{T})})$, where $\mathcal{T}$ is a tree and each node x of $\mathcal{T}$ is associated with a subset of vertices $B_x \subseteq V(G)$, called the *bag at* x. Moreover, the following condition have to be satisfied:

- The sets $\{B_x\}_{x \in V(\mathcal{T})}$ form a partition of $V(G)$, and in particular are pairwise disjoint.
- For each edge $uv \in E(G)$, either there is some $x \in V(\mathcal{T})$ such that $\{u, v\} \subseteq B_x$, or there is some $xy \in E(\mathcal{T})$ such that $u \in B_x$ and $v \in B_y$.

The *width* of $\mathcal{T}$ is equal to $\max_{x \in V(\mathcal{T})} |B_x|$, and the *tree-partition width of* G, denoted $\mathtt{tpw}(G)$, is the minimum possible width of a tree-partition decomposition of G.

It is easy to see that empty bags in a tree-partition decomposition can be disposed of, implying $|V(\mathcal{T})| \leq |V(G)|$ without loss of generality. For all G, $1 + \mathtt{tw}(G) \leq 2 \cdot \mathtt{tpw}(G)$, but $\mathtt{tpw}(G)$ can be arbitrarily large already for graphs of constant treewidth, unless the maximum degree is bounded [65]. We also need the following well-known fact.

Lemma 2.6 (cf. Exercise 7.15 in [21])**.** *A graph on n vertices of treewidth k has at most kn edges. Hence a graph on n vertices of tree-partition width k has at most $2kn$ edges.*

Measures and balanced separators. In several parts of the paper, we will be introducing auxiliary weight functions on the vertices of graphs, which we call *measures.*

Definition 2.7. Let G be a graph. Any function $\mu \colon V(G) \to \mathbb{R}^+ \cup \{0\}$ that is positive on at least one vertex is called a *measure on* $V(G)$. For a subset $A \subseteq V(G)$, we denote $\mu(A) = \sum_{u \in A} \mu(u)$.

First, we need the following simple folklore lemma about the existence of balanced nodes of trees.

Lemma 2.8. *Let T be a tree on q nodes, with a measure μ defined on $V(T)$. Then a node $x \in V(T)$ such that $\mu(V(C)) \leq \mu(V(T))/2$ for every $C \in \mathtt{cc}(T - x)$ can be found in time $\mathcal{O}(q)$.*

Proof. Consider any edge $yz \in E(T)$, and let the removal of yz split T into subtrees T_y and T_z, where $y \in V(T_y)$ and $z \in V(T_z)$. Orient yz from y to z if $\mu(V(T_y)) < \mu(V(T_z))$, from z to y if $\mu(V(T_y)) > \mu(V(T_z))$, and arbitrarily if $\mu(V(T_y)) = \mu(V(T_z))$. The obtained oriented tree has q nodes and $q - 1$ directed edges, which means that there is a node x that has indegree 0. For every neighbor y of x we have that the edge xy was directed towards x. This means that $\mu(V(T_x)) \geq \mu(V(T_y))$; equivalently $\mu(V(T_y)) \leq \mu(V(T))/2$. As y was an arbitrarily chosen neighbor of x, it follows that x satisfies the required property.

As for the algorithmic claim, it is easy to implement the procedure orienting the edges in time $\mathcal{O}(q)$ by using a recursive depth-first search procedure on T that returns the total weight of nodes in the explored subtree. Having the orientation computed, suitable x can be retrieved by a simple indegree count in time $\mathcal{O}(q)$. $\square$

In graphs of bounded treewidth, Lemma 2.8 can be generalized to find balanced bags instead of balanced nodes.

Definition 2.9. Let G be a graph, let μ be a measure on $V(G)$, and let $\alpha \in [0, 1]$. A set $X \subseteq V(G)$ is called an *α-balanced separator w.r.t. μ* if for each $C \in \mathtt{cc}(G - X)$, it holds that $\mu(V(C)) \leq \alpha \cdot \mu(V(G))$.

Lemma 2.10 (Lemma 7.19 of [21])**.** *Let G be a graph with $\mathtt{tw}(G) < k$, and let μ be a measure on $V(G)$. Then there exists a $\frac{1}{2}$-balanced separator X w.r.t. μ with $|X| \leq k$.*

3 Approximating treewidth

In this section we show our approximation algorithm for treewidth, i.e., prove Theorem 1.1. For the reader's convenience, we restate it here.

Theorem 1.1. *There exists an algorithm that, given a graph G on n vertices and a positive integer k, in time $\mathcal{O}(k^7 \cdot n \log n)$ either provides a tree decomposition of G of width at most $\mathcal{O}(k^2)$, or correctly concludes that $\mathtt{tw}(G) \geq k$.*

In our proof of Theorem 1.1, we obtain an upper bound of $1800k^2$ on the width of the computed tree decomposition. We remark that this number can be improved by a more careful analysis of different parameters used throughout the algorithm; however, we refrain from performing a tighter analysis in order to simplify the presentation.

We first prove the backbone technical result, that is, an approximation algorithm for finding balanced separators. This algorithm will be used as a subroutine in every step of the algorithm of Theorem 1.1. Our approach for approximating balanced separators is based on the work of Feige and Mahdian in [28].

Lemma 3.1. *There exists an algorithm that, given a graph G on n vertices and m edges with a measure μ on $V(G)$, and a positive integer k, works in time $\mathcal{O}(k^4 \cdot (n+m))$ and returns one of the following outcomes:*

(1) A $\frac{7}{8}$-balanced separator Y w.r.t. μ with $|Y| \leq 100k^2$;

(2) A $(1 - \frac{1}{100k})$-balanced separator X w.r.t. μ with $|X| \leq k$;

(3) A correct conclusion that $\mathtt{tw}(G) \geq k$.

Proof. By rescaling μ if necessary, we assume that $\mu(V(G)) = 1$. Throughout the proof we assume that $\mathtt{tw}(G) < k$ and hence, by Lemma 2.10, there exists some $\frac{1}{2}$-balanced separator W w.r.t. μ, which is of course unknown to the algorithm. We will prove that whenever such a W exists, the algorithm reaches one of the outcomes (1) or (2). If none of these outcomes is reached, then no such W exists and, by Lemma 2.10, the algorithm can safely report that $\mathtt{tw}(G) \geq k$, i.e., reach outcome (3). We also assume that for every vertex $u \in V(G)$ it holds that $\mu(u) < \frac{1}{100k}$, because otherwise we can immediately provide outcome (2) by setting $X = \{u\}$. Note that in particular this implies that $\mu(W) < \frac{1}{100}$.

We first generalize the problem slightly. Suppose that for some $i \leq k$ we are given a set Y_i such that the following invariants are satisfied: (i) $|Y_i| \leq 100ik$ and (ii) $|W \cap Y_i| \geq i$. Then, the claim is as follows:

Claim 3.2. *Given Y_i satisfying invariants (i) and (ii) for some $i < k$, one can in time $\mathcal{O}(k^3 \cdot (n+m))$ either arrive at one of the outcomes* (1) *or* (2)*, or find a set Z with $|Z| \leq 100k$ and $Z \cap Y_i = \emptyset$, such that $Z \cap W \neq \emptyset$.*

Before we proceed to the proof of Claim 3.2, we observe how Lemma 3.1 follows from it. We start with $Y_0 = \emptyset$, which clearly satisfies the invariants (i) and (ii). Then we iteratively compute $Y_1, Y_2, Y_3, \ldots$ as follows: when computing Y_{i+1}, we use the algorithm of Claim 3.2 to either provide outcome (1) or (2), in which case we terminate the whole computation, or find a suitable set Z. Then $Y_{i+1} = Y_i \cup Z$ satisfies the invariants (i) and (ii) for the next iteration, and hence we can proceed. Suppose that this algorithm successfully performed k iterations, i.e., it constructed Y_k. Then we have that $|W \cap Y_k| \geq k$, so since $|W| \leq k$, we have $W \subseteq Y_k$. Then Y_k should be a $\frac{1}{2}$-balanced separator w.r.t. μ and $|Y_k| \leq 100k^2$, so it can be reported as Y in outcome (1). If Y_k is not a $\frac{1}{2}$-balanced separator, then W did not exist in the first place and the algorithm can safely report outcome (3). Since the algorithm of Claim 3.2 works in time $\mathcal{O}(k^3 \cdot (n+m))$ and we apply it at most k times, the running time promised in the lemma statement follows.

We now proceed to the proof of Claim 3.2. Let $G' = G - Y_i$. First, let us investigate the connected components of G'. If $\mu(V(C)) \leq \frac{7}{8}$ for each $C \in \mathtt{cc}(G')$, then we can reach outcome (1) by taking $Y = Y_i$, because $|Y_i| \leq 100ik \leq 100k^2$. Hence, suppose there is a connected component $C_0 \in \mathtt{cc}(G')$ with $\mu(C_0) > \frac{7}{8}$. Let T_0 be an arbitrary spanning tree of C_0, and let $V_0 = V(C_0)$. We first prove an auxiliary claim that will imply that we can find a nice partitioning of T_0. This is almost exactly the notion of *Steiner decompositions* used by Feige and Mahdian [28] (see Definition 5.1 and Lemma 5.2 in [28]), but we choose to reprove the result for the sake of completeness.

Claim 3.3. *Suppose we are given a number $\lambda \in \mathbb{R}^+$ and a tree T on n vertices with a measure μ on $V(T)$, such that $\mu(u) < \lambda$ for each $u \in V(T)$. Then one can in time $\mathcal{O}(n)$ find a family $\mathcal{F} = \{(R_1, u_1), (R_2, u_2), \ldots, (R_p, u_p)\}$ (with u_is not necessarily distinct) such that the following holds (in the following, we denote $\tilde{R}_i = R_i \setminus \{u_i\}$):*

(a) for each $i = 1, 2, \ldots, p$, we have that $u_i \in R_i \subseteq V(T)$, $T[R_i]$ is connected, and $\lambda \leq \mu(\tilde{R}_i) < 4\lambda$;

(b) $\mu(V(T) \setminus \bigcup_{i=1}^{p} \tilde{R}_i) < 2\lambda$;

(c) sets $\tilde{R}_i$ are pairwise disjoint for $i = 1, 2, \ldots, p$.

Proof. We first provide a combinatorial proof of the existence of $\mathcal{F}$, which proceeds by induction on n. Then we will argue how the proof gives rise to an algorithm constructing $\mathcal{F}$ in linear time.

Let us root T in an arbitrary vertex r, which imposes a parent-child relation on the vertices of T. For $v \in V(T)$, let T_v be the subtree rooted at v. If $\mu(V(T)) < 2\lambda$ then we can take $\mathcal{F} = \emptyset$, so suppose otherwise. Let then u be the deepest vertex of T for which $\mu(V(T_u)) \geq 2\lambda$, and let $u_1, u_2, \ldots, u_q$ be the children of u. Let us denote $a_j = \mu(V(T_{u_j}))$, for $j = 1, 2, \ldots, q$. As u was chosen to be the deepest, we have that $a_j < 2\lambda$ for each $j = 1, 2, \ldots, q$. Moreover, since $\mu(u) < 2\lambda$ and $\mu(V(T_u)) \geq 2\lambda$, we have that u has at least one child, i.e., $q > 0$.

Scan the sequence $a_1, a_2, \ldots, a_q$ from left to right, and during this scan iteratively extract minimal prefixes for which the sum of entries is at least λ, up to the point when the total sum of the remaining numbers is smaller than λ. Let these extracted sequences be

$$\{a_1, a_2, \ldots, a_{j_2-1}\}, \{a_{j_2}, a_{j_2+1}, \ldots, a_{j_3-1}\}, \ldots, \{a_{j_s}, a_{j_s+1}, \ldots, a_{j_{s+1}-1}\},$$

where s is their number. Then, denoting $a_{j_1} = 1$, we infer from the construction and the fact that $a_j < 2\lambda$ for all j that the following assertions hold:

- $\lambda \leq \sum_{i=j_\ell}^{j_{\ell+1}-1} a_i < 3\lambda$ for all $\ell = 1, 2, \ldots, s$; and
- $\sum_{i=j_{s+1}}^{q} a_i < \lambda$.

Moreover, since $\mu(V(T_u)) \geq 2\lambda$, we have that $\sum_{i=1}^{q} a_i \geq 2\lambda - \mu(u) > \lambda$, and hence at least one sequence has been extracted; i.e., $s \geq 1$.

For $\ell = 1, \ldots, s-1$, let $I_\ell = \{j_\ell, j_\ell + 1, \ldots, j_{\ell+1} - 1\}$, and let $I_s = \{j_s, j_s + 1, \ldots, q\}$. Concluding, from the assertions above it follows that we have partitioned $\{1, \ldots, q\}$ into contiguous sets of indices $I_1, I_2, \ldots, I_s$ such that $s \geq 1$ and $\lambda \leq \sum_{i \in I_\ell} a_i < 4\lambda$, for each $\ell = 1, 2, \ldots, s$.

Let T' be T with all the subtrees T_{u_j} removed, for $j = 1, 2, \ldots, q$. Since $q > 0$, we have that $|V(T')| < |V(T)|$, and hence by the induction hypothesis we can find a family $\mathcal{F}' = \{(R_1, u_1), (R_2, u_2), \ldots, (R_{p'}, u_{p'})\}$ for the tree T' satisfying conditions (a), (b) and (c).

For $\ell = p'+1, p'+2, \dots, p'+s$, let $R_{p'+\ell} = \{u\} \cup \bigcup_{i \in I_\ell} V(T_{u_i})$ and $u_\ell = u$. Observe that $\mu(R_{p'+\ell} \setminus \{u_{p'+\ell}\}) = \sum_{i \in I_\ell} a_i$, so we obtain that $\lambda \le \mu(\tilde{R}_{p'+\ell}) < 4\lambda$. Consider $\mathcal{F} = \mathcal{F}' \cup \{(R_{p'+1}, u_{p'+1}), (R_{p'+2}, u_{p'+2}), \dots, (R_{p'+s}, u_{p'+s})\}$. It can be easily verified that conditions (a), (b), and (c) hold for $\mathcal{F}$ using the induction assumption that they held for $\mathcal{F}'$. This concludes the inductive proof of the existence of $\mathcal{F}$.

Naively, the proof presented above gives rise to an $\mathcal{O}(n^2)$ algorithm that iteratively finds a suitable vertex u, and applies itself recursively to T'. However, it is very easy to see that the algorithm can be implemented in time $\mathcal{O}(n)$ by processing T bottom-up, remembering the total measure of the vertices in the processed subtree, and cutting new pairs (R_i, u_i) whenever the accumulated measure exceeds 2λ. $\lrcorner$

Armed with Claim 3.3, we proceed with the proof of Claim 3.2. Apply the algorithm of Claim 3.3 to tree T_0 and $\lambda = \frac{1}{100k}$. Note that the premise of the claim holds by the assumption that $\mu(u) < \frac{1}{100k}$ for each $u \in V(G)$. Therefore, we obtain a family $\mathcal{F} = \{(R_1, u_1), (R_2, u_2), \dots, (R_p, u_p)\}$ satisfying conditions (a), (b), and (c) for T_0. For $i = 1, 2, \dots, p$, let $\tilde{R}_i = R_i \setminus \{u_i\}$ and let $Z = \{u_1, u_2, \dots, u_p\}$. Note that by conditions (a) and (c) we have that

$$1 = \mu(V(G)) \ge \sum_{i=1}^{p} \mu(\tilde{R}_i) \ge p\lambda = \frac{p}{100k},$$

so $|Z| \le p \le 100k$.

We need the following known fact.

Claim 3.4 (cf. the proof of Lemma 7.20 in [21])**.** *Let $b_1, b_2, \dots, b_q$ be nonnegative reals such that $\sum_{i=1}^{q} b_i \le 1$ and $b_i \le 1/2$ for each $i = 1, 2, \dots, q$. Then $\{1, \dots, q\}$ can be partitioned into two sets J_1 and J_2 such that $\sum_{i \in J_z} b_i \le \frac{2}{3}$ for each $z \in \{1, 2\}$.*

Let $B_1, B_2, \dots, B_q$ be the connected components of $C_0 - W$, and let $b_i = \mu(V(B_i))$ for each $i = 1, 2, \dots, q$. Clearly $\sum_{i=1}^{q} b_i \le \mu(V(G)) = 1$. Moreover, each component B_i is contained in some component of $G - W$, and hence, since W is a $\frac{1}{2}$-balanced separator of G w.r.t. measure μ, we have that $b_i \le 1/2$ for each $i = 1, 2, \dots, q$. By Claim 3.4 we can find a partition (J_1, J_2) of $\{1, \dots, q\}$ such that $\sum_{i \in J_z} b_i \le \frac{2}{3}$ for each $z \in \{1, 2\}$. Let $A_1 = \bigcup_{i \in J_1} V(B_i)$ and $A_2 = \bigcup_{i \in J_2} V(B_i)$. Then vertex sets A_1 and A_2 are non-adjacent in G and $\mu(A_1), \mu(A_2) \le \frac{2}{3}$. Recall that $\frac{7}{8} < \mu(V(C_0)) = \mu(A_1) + \mu(A_2) + \mu(W \cap V(C_0))$. Since $\mu(W) < \frac{1}{100}$, we have that $\mu(A_1) + \mu(A_2) > \frac{7}{8} - \frac{1}{100} > \frac{5}{6}$. Hence it follows that $\mu(A_1), \mu(A_2) > \frac{5}{6} - \frac{2}{3} = \frac{1}{6}$.

Let $K_1 \subseteq \{1, \dots, p\}$ be the set of those indices i for which $\tilde{R}_i \cap A_1 \ne \emptyset$. By properties (a), (b), and (c), we obtain that:

$$\frac{1}{6} < \mu(A_1) \le \mu\left(V(C_0) \setminus \bigcup_{i=1}^{p} \tilde{R}_i\right) + \sum_{i \in K_1} \mu(\tilde{R}_i) < 2\lambda + 4\lambda|K_1|.$$

Since $\lambda = \frac{1}{100k}$, we have that

$$|K_1| > \frac{1}{24\lambda} - \frac{1}{2} > 3k.$$

Since $|W| \le k$ and sets $\tilde{R}_i$ are pairwise disjoint, there is $L_1 \subseteq K_1$ of size at least $2k$ such that additionally $\tilde{R}_i \cap W = \emptyset$ for each $i \in L_1$. Symmetrically we prove that there is a set $L_2 \subseteq \{1, \dots, p\}$ of at least $2k$ indices such that $\tilde{R}_i \cap A_2 \ne \emptyset$ and $\tilde{R}_i \cap W = \emptyset$, for each $i \in L_2$.

Suppose for a moment that $Z \cap W = \emptyset$. Then, for each $i \in L_1$ we in fact have that $R_i \cap W = \emptyset$. Since $G[R_i]$ is connected, $\tilde{R}_i \cap A_1 \neq \emptyset$, and there is no edge between A_1 and A_2, it follows that $R_i \subseteq A_1$. Similarly, $R_i \subseteq A_2$ for each $i \in L_2$.

Therefore, the algorithm does as follows. For each pair of distinct indices $i, j \in \{1, \ldots, p\}$ for which $R_i \cap R_j = \emptyset$, we verify whether the size of a minimum vertex cut in G between R_i and R_j does not exceed k. If we find such a pair and the corresponding vertex cut X, then X separates R_i from R_j, so X is a $(1 - \frac{1}{100k})$-balanced separator w.r.t. μ in G, due to $\mu(R_i), \mu(R_j) \geq \frac{1}{100k}$. Therefore, such X can be reported as outcome (2). The argumentation of the previous paragraph ensures us that at least one such pair (i, j) will be found provided $Z \cap W = \emptyset$.

Hence, if for every such pair (i, j) the minimum vertex cut in G between R_i and R_j is larger than k, then we have a guarantee that $Z \cap W \neq \emptyset$. Since $|Z| \leq 100k$, we can provide the set Z as the outcome of the algorithm of Claim 3.2. This concludes the description of the algorithm of Claim 3.2. To bound its running time, observe that the application of the algorithm of Claim 3.3 takes time $\mathcal{O}(n + m)$, whereas later we verify the minimum value of a vertex cut for at most $p^2 = \mathcal{O}(k^2)$ pairs (i, j). By Corollary 2.2, each such verification can be implemented in time $\mathcal{O}(k \cdot (n + m))$. Hence, the whole algorithm of Claim 3.2 indeed runs in time $\mathcal{O}(k^3 \cdot (n + m))$. As argued before, Lemma 3.1 follows from Claim 3.2. $\square$

Having an approximation algorithm for balanced separators, we can proceed with the proof of Theorem 1.1. Following the approach introduced by Robertson and Seymour [57], we solve a more general problem. Let $\eta = 100k^2$. Assume we are given a graph H together with a subset $S \subseteq V(H)$, $S \neq V(H)$, with the invariant that $|S| \leq 17\eta$. The goal is to either conclude that $\mathtt{tw}(H) \geq k$, or to compute a rooted tree decomposition of H, i.e., a tree decomposition rooted at some node r, that has width at most 18η and where S is a subset of the root bag. Note that if we find an algorithm with running time $\mathcal{O}(k^7 \cdot n \log n)$ for the generalized problem, then Theorem 1.1 will follow by applying it to G and $S = \emptyset$.

The algorithm for the more general problem works as follows. First, observe that we can assume that $|E(H)| \leq kn$, where we denote $n = |V(H)|$. Indeed, by Lemma 2.6 we can immediately answer that $\mathtt{tw}(H) \geq k$ if this assertion does not hold. Having this assumption, we consider three cases: either (i) $|V(H) \setminus S| \leq \eta$, or, if this case does not hold, (ii) $|S| \leq 16\eta$, or (iii) $16\eta < |S| \leq 17\eta$.

Case (i): If $|V(H) \setminus S| \leq \eta$, we can output a trivial tree decomposition with one bag containing $V(H)$, because the facts that $|V(H) \setminus S| \leq \eta$ and $|S| \leq 17\eta$ imply that $|V(H)| \leq 18\eta$. In the other cases we assume that (i) does not hold, i.e., $|V(H) \setminus S| > \eta$.

Case (ii): Suppose $|S| \leq 16\eta$. Define a measure $\mu_{\overline{S}}$ on $V(H)$ as follows: $\mu_{\overline{S}}(u) = 0$ for each $u \in S$, and $\mu_{\overline{S}}(u) = 1$ for each $u \notin S$. Run the algorithm of Lemma 3.1 on H with measure $\mu_{\overline{S}}$. If it concluded that $\mathtt{tw}(H) \geq k$, then we can terminate and pass this answer. Otherwise, let X be the obtained subset of vertices. Regardless whether outcome (1) or (2) was given, we have that X is a $(1 - \frac{1}{100k})$-balanced separator in H w.r.t. measure $\mu_{\overline{S}}$, and moreover $|X| \leq \eta$. Let $B = S \cup X$ and observe that $|B| \leq |S| + |X| \leq 17\eta$. Consider the connected components of $H - B$. For each $C \in \mathtt{cc}(H - B)$, define a new instance (H_C, S_C) of the generalized problem as follows: $H_C = H[N_H[V(C)]]$ and $S_C = N_H(V(C))$. Observe that $S_C \subseteq B$, hence $|S_C| \leq |B| \leq 17\eta$, and thus the invariant that $|S_C| \leq 17\eta$ is satisfied in the instance (H_C, S_C). Apply the algorithm recursively to (H_C, S_C), yielding either a conclusion that $\mathtt{tw}(H_C) \geq k$, in which case we can report that $\mathtt{tw}(H) \geq k$ as well, or a rooted tree decomposition $\mathcal{T}_C$ of H_C that has width at most 18η and where S_C is a subset of the root bag. Construct a tree decomposition $\mathcal{T}$ of H as follows: create a root node r associated with bag B, and, for each $C \in \mathtt{cc}(H - B)$, attach the decomposition $\mathcal{T}_C$

below r by making the root of $\mathcal{T}_C$ a child of r. It easy to verify that $\mathcal{T}$ created in this manner indeed is a tree decomposition of H, and its width does not exceed 18η because $|B| \leq 17\eta \leq 18\eta$.

Case (iii): Suppose $16\eta < |S| \leq 17\eta$. Define a measure μ_S on $V(H)$ as follows: $\mu_S(u) = 1$ for each $u \in S$, and $\mu_S(u) = 0$ for each $u \notin S$. Run the algorithm of Lemma 3.1 on H with measure μ_S. Again, if it concluded that $\mathtt{tw}(H) \geq k$, then we can terminate and pass this answer. Otherwise, we obtain either a $\frac{7}{8}$-balanced separator Y with $|Y| \leq \eta$, or a $(1 - \frac{1}{100k})$-balanced separator X with $|X| \leq k$. Let Z be this separator, being either X or Y depending on the subcase. The algorithm proceeds as in the previous case. Define $B = S \cup Z$; then $|B| \leq |S| + |Z| \leq 18\eta$. For each $C \in \mathtt{cc}(H - B)$, define a new instance (H_C, S_C) of the generalized problem by taking $(H_C, S_C) = (H[N_H[V(C)]], N_H(V(C)))$. Apply the algorithm recursively to each (H_C, S_C), yielding either a conclusion that $\mathtt{tw}(H_C) \geq k$, which implies $\mathtt{tw}(H) \geq k$, or a tree decomposition $\mathcal{T}_C$ of H_C that has width at most 18η and S_C is contained in its root bag. Construct the output decomposition $\mathcal{T}$ by taking B as the bag of the root node r, and attaching all the decompositions $\mathcal{T}_C$ below r by making their roots children of r. Again, it can be easily verified that $\mathcal{T}$ is a tree decomposition of H of width at most 18η. The only verification that was not performed is that in the new instances (H_C, S_C), the invariant that $|S_C| \leq 17\eta$ still holds. We shall prove an even stronger fact: that $|S_C| \leq |S| - k$, for each $C \in \mathtt{cc}(H - B)$, so the needed invariant will follow by $|S| \leq 17\eta$. However, we will need this stronger property in the future. The proof investigates the subcases $Z = X$ and $Z = Y$ separately.

Suppose first that $Z = X$, that is, the algorithm of Lemma 3.1 have found a $(1 - \frac{1}{100k})$-balanced separator X with $|X| \leq k$. Consider any connected component $C \in \mathtt{cc}(H - B)$, and let D be the connected component of $H - X$ in which C is contained. Since X is $(1 - \frac{1}{100k})$-balanced w.r.t. μ_S, we have that

$$|S \cap D| = \mu_S(D) \leq \left(1 - \frac{1}{100k}\right) \cdot \mu_S(V(H)) = \left(1 - \frac{1}{100k}\right) \cdot |S| = |S| - \frac{|S|}{100k} < |S| - 16k;$$

the last inequality follows from the assumption that $|S| > 16\eta = 1600k^2$. Now observe that $N_H(C) \subseteq X \cup (S \cap D)$, so $|N_H(C)| < k + (|S| - 16k) = |S| - 15k < |S| - k$.

Suppose second that $Z = Y$, that is, the algorithm of Lemma 3.1 found a $\frac{7}{8}$-balanced separator Y with $|Y| \leq \eta$. Again, consider any connected component $C \in \mathtt{cc}(H - B)$, and let D be the connected component of $H - Y$ in which C is contained. Since Y is $\frac{7}{8}$-balanced w.r.t. μ_S, we have that

$$|S \cap D| = \mu_S(D) \leq \frac{7}{8} \cdot \mu_S(V(H)) = \frac{7}{8}|S| = |S| - \frac{|S|}{8} < |S| - 2\eta;$$

the last inequality follows from the assumption that $|S| > 16\eta$. Again, observe that $N_H(C) \subseteq Y \cup (S \cap D)$, so $|N_H(C)| < \eta + (|S| - 2\eta) = |S| - \eta < |S| - k$.

This concludes the description of the algorithm. Its partial correctness is clear: if the algorithm concludes that $\mathtt{tw}(H) \geq k$, then it is always correct, and similarly any tree decomposition of H output by the algorithm satisfies the specification. We are left with arguing that the algorithm always stops (i.e., it does not loop in the recursion) and its running time is bounded by $\mathcal{O}(k^7 \cdot n \log n)$. We argue both these properties simultaneously.

Let $\mathfrak{T}$ be the recursion tree yielded by the algorithm. That is, $\mathfrak{T}$ is a rooted tree with nodes corresponding to recursive subcalls to the algorithm (in case the algorithm loops, $\mathfrak{T}$ would be infinite). Thus, each node is labeled by the instance (H', S') which is being solved in the subcall. The root of $\mathfrak{T}$ corresponds to the original instance (H, S), and the children of each node x correspond to subcalls invoked in the call corresponding to x. Observe that if the algorithm returns some decomposition $\mathcal{T}$ of H, then $\mathcal{T}$ is isomorphic to $\mathfrak{T}$, because every subcall produces exactly one new bag, and the bags are arranged into $\mathcal{T}$ exactly according to the recursion tree of the algorithm. The leaves of $\mathfrak{T}$ correspond to calls where no subcall was invoked: either ones conforming to Case (i), or ones conforming to Case (ii) or (iii) when $B = V(H)$. Partition the node set of $\mathfrak{T}$ into sets $A_{\text{(i)}}$, $A_{\text{(ii)}}$, $A_{\text{(iii)}}$, depending whether the corresponding subcall falls into Case (i), (ii), or (iii). For each node $x \in V(\mathfrak{T})$, let (H_x, S_x) be its corresponding subcall, let $h_x = |V(H_x) \setminus S_x|$, $s_x = |S_x|$, and $n_x = h_x + s_x = |V(H_x)|$. By $\mathtt{chld}(x)$ we denote the set of children of x. The following claim shows the crucial properties of $\mathfrak{T}$ that follow from the algorithm.

Claim 3.5. *The following holds:*

(a) For each $x \in A_{(ii)}$ and each $y \in \mathtt{chld}(x)$, we have $h_y \leq (1 - \frac{1}{100k}) \cdot h_x$.

(b) For each $x \in A_{(iii)}$ and each $y \in \mathtt{chld}(x)$, we have $s_y \leq s_x - k$.

Proof. Assertion (b) is exactly the stronger property $|N_H(C)| \leq |S| - k$ that we have ensured in Case (iii). Hence, it remains to prove assertion (a). Let $X \subseteq V(H_x)$ be the separator found by the algorithm when investigating the subcall in node x. Suppose that child y corresponds to some subcall $(H_y, S_y) = (H_C, S_C) = (N_{H_x}[V(C)], N_{H_x}(V(C)))$, for some component $C \in \mathtt{cc}(H_x - (S_x \cup X))$. Let D be the connected component of $H - X$ in which C is contained. Since X is $(1 - \frac{1}{100k})$-balanced in H_x w.r.t. measure $\mu_{\overline{S}}$, we have that

$$h_y = |C| \leq |D \setminus S| = \mu_{\overline{S}}(D) \leq \left(1 - \frac{1}{100k}\right) \cdot \mu_{\overline{S}}(V(H_x)) = \left(1 - \frac{1}{100k}\right) \cdot h_x.$$

⌟

Using Claim 3.5, we can show an upper bound on the depth of $\mathfrak{T}$. This in particular proves that the algorithm always stops (equivalently, $\mathfrak{T}$ is finite).

Claim 3.6. *The depth of $\mathfrak{T}$ is bounded by $\mathcal{O}(k^2 \cdot \log n)$.*

Proof. Take any finite path P in $\mathfrak{T}$ that starts in its root and travels down the tree. Since each node of $A_{\text{(i)}}$ is a leaf in $\mathfrak{T}$, at most one node of P can belong to $A_{\text{(i)}}$. We now estimate how many nodes of $A_{\text{(ii)}}$ and $A_{\text{(iii)}}$ can appear on P.

First, we claim that there are at most $\mathcal{O}(k \cdot \log n)$ nodes of $A_{\text{(ii)}}$ on P. Let $x_0, x_1, x_2, \ldots, x_p$ be consecutive vertices of $A_{\text{(ii)}}$ on P, ordered from the root. By Claim 3.5(a) we have that $h_{x_{i+1}} \leq (1 - \frac{1}{100k}) \cdot h_{x_i}$ for each nonnegative integer i. Since $h_{x_0} \leq n$, by induction it follows that $h_{x_i} \leq (1 - \frac{1}{100k})^i \cdot n$. However, $h_x \geq 1$ for each $x \in V(\mathfrak{T})$, so $p \leq \log_{1-\frac{1}{100k}} 1/n \leq \mathcal{O}(k \cdot \log n)$. Consequently, $|V(P) \cap A_{\text{(ii)}}| = p + 1 \leq \mathcal{O}(k \cdot \log n)$.

Second, we claim that on P there cannot be more than $100k$ consecutive vertices from $A_{\text{(iii)}}$. Assume otherwise, that there exists a sequence $x_0, x_1, \ldots, x_{100k}$ where $x_i \in V(P) \cap A_{\text{(iii)}}$ for each $0 \leq i \leq 100k$ and $x_{i+1} \in \mathtt{chld}(x_i)$ for each $0 \leq i < 100k$. By Claim 3.5(b) we have that $s_{x_{i+1}} \leq s_{x_i} - k$, for each $0 \leq i < 100k$. By induction we obtain that $s_{x_i} \leq s_{x_0} - ik$. However, $s_{x_0} \leq 17\eta$, so $s_{x_{100k}} \leq 17\eta - 100k \cdot k = 16\eta$. This is a contradiction with $x_{100k} \in A_{\text{(iii)}}$, because Case (ii) should apply to instance $(H_{x_{100k}}, S_{x_{100k}})$ instead.

Combining the observations of the previous paragraphs, we see that P has at most $\mathcal{O}(k \cdot \log n)$ vertices of $A_{\text{(ii)}}$, the segments of consecutive vertices of $A_{\text{(iii)}}$ have length at most $100k$, and only the last vertex can belong to $A_{\text{(i)}}$. It follows that $|V(P)| \leq \mathcal{O}(k^2 \cdot \log n)$. Since P was chosen arbitrarily, we infer that the depth of $\mathfrak{T}$ is bounded by $\mathcal{O}(k^2 \cdot \log n)$. ⌟

By Claim 3.6, we already know that the algorithm stops and outputs some tree decomposition $\mathcal{T}$. As we noted before, $\mathcal{T}$ and $\mathfrak{T}$ are isomorphic by a mapping that associates a bag of $\mathcal{T}$ with the subcall in which it was constructed. By somewhat abusing the notation, we will identify $\mathcal{T}$ with $\mathfrak{T}$ in the sequel. It remains to argue that the running time of the algorithm is bounded by $\mathcal{O}(k^7 \cdot n \log n)$.

We partition the total work used by the algorithm between the nodes of $\mathcal{T}$. For $x \in V(\mathcal{T})$, we assign to x the operations performed by the algorithm when constructing the bag B_x associated with x: running the algorithm of Lemma 3.1 to compute an appropriate balanced separator, construction of the subcalls, and gluing the subdecompositions obtained from the subcalls below the constructed bag B_x. From the description of the algorithm and Lemma 3.1 it readily follows that the work associated with node $x \in V(\mathcal{T})$ is bounded by $\mathcal{O}(n_x)$ whenever x belongs $A_{\text{(i)}}$, and by $\mathcal{O}(k^5 \cdot n_x)$ whenever x belongs to $A_{\text{(ii)}}$ or $A_{\text{(iii)}}$. Note that here we use the assumption that $|E(H_x)| \leq k \cdot n_x$ for each node $x \in V(\mathcal{T})$; we could assume this because otherwise the application of the algorithm to instance (H_x, S_x) would immediately reveal that $\mathtt{tw}(H_x) \geq k$.

Take any node $x \in A_{\text{(i)}}$, and observe that, since $S_x \neq V(H_x)$, we have that $B_x = V(H_x)$ contains some vertex of H which does not belong to any other bag of $\mathcal{T}$: any vertex of $V(H_x) \setminus S_x$ has this property. Therefore, the total number of nodes in $A_{\text{(i)}}$ is at most n. Since $n_x \leq \mathcal{O}(k^2)$ for each such x, we infer that the total work associated with bags of $A_{\text{(i)}}$ is bounded by $\mathcal{O}(k^2 \cdot n)$.

We now bound the work associated with the nodes of $A_{\text{(ii)}} \cup A_{\text{(iii)}}$. Let d be the depth of $\mathcal{T}$; by Claim 3.6 we know that $d \leq \mathcal{O}(k^2 \cdot \log n)$. For $0 \leq i \leq d$, let L_i be the set of nodes of $A_{\text{(ii)}} \cup A_{\text{(iii)}}$ that are at depth exactly i in $\mathcal{T}$. Fix some i with $0 \leq i \leq d$. Observe that among $x \in L_i$, the sets $V(H_x) \setminus S_x$ are pairwise disjoint. Hence, $\sum_{x \in L_i} h_x \leq n$. However, for every node $x \in A_{\text{(ii)}} \cup A_{\text{(iii)}}$ we have that $h_x > \eta$ (or otherwise case (i) would apply in the subcall corresponding to x) and $s_x \leq 17\eta$. This implies that $n_x = h_x + s_x \leq 18 h_x$, and hence $\sum_{x \in L_i} n_x \leq 18n$. Consequently, the total work associated with the nodes of L_i is bounded by $\sum_{x \in L_i} \mathcal{O}(k^5 \cdot n_x) \leq \mathcal{O}(k^5 \cdot n)$. Since $d = \mathcal{O}(k^2 \log n)$, we conclude that the total work associated with bags of $A_{\text{(ii)}} \cup A_{\text{(iii)}}$ is $\mathcal{O}(k^7 \cdot n \log n)$.

Concluding, we have shown that the algorithm for the generalized problem is correct, always terminates, and in total uses time $\mathcal{O}(k^7 \cdot n \log n)$. Theorem 1.1 follows by applying it to $H = G$ and $S = \emptyset$.

4 Gaussian elimination

First, in Section 4.1 we describe our Gaussian elimination algorithm guided by an ordering of rows and columns of the matrix. Next, in Section 4.2 we show how suitable orderings of low width can be recovered from low-width path and tree-partition decompositions of the bipartite graph associated with the matrix. Finally, in Section 4.3 we present how the approach can be lifted to tree decompositions using an adaptation of the vertex-splitting/matrix sparsification technique.

4.1 Gaussian elimination with strong orderings

Algebraic description. We first describe our algorithm purely algebraically, showing what arithmetic operations will be performed and in which order. We will address implementation details later.

We assume that the algorithm is given an $n \times m$ matrix M over some field $\mathbb{F}$, and moreover there is an ordering $\preccurlyeq$ imposed on the rows and columns of M. We will never compare rows with columns using $\preccurlyeq$, so actually we are only interested in the orders imposed by $\preccurlyeq$ on the rows and on the columns of M. The following Algorithm 1 presents our procedure.

Input: An $n \times m$ matrix M with an order $\preccurlyeq$ on rows and columns of M
Output: Matrices U, L, and removal orders of rows and columns of M

```
Set $\mathcal{I}$, $\mathcal{J}$ to be the sets of all columns and all rows of $M$, respectively.
Set $L$ to be the $n \times n$ identity matrix.
while $\mathcal{I}$ is not empty do
    Let $i$ be the earliest column in $\mathcal{I}$ (in the ordering $\preccurlyeq$);
    if it is empty, remove $i$ from $\mathcal{I}$ and choose $i$ again.
    Let $j$ be the earliest row in $\mathcal{J}$ such that $M[j,i] \neq 0$ (in the ordering $\preccurlyeq$).
    for every row $k \neq j$ in $\mathcal{J}$ such that $M[k,i] \neq 0$ do
        /* Eliminate entry $M[k,i]$ using row $j$. */
        Set $L[k,j] := \frac{M[k,i]}{M[j,i]}$.
        for every column $\ell$ such that $M[j,\ell] \neq 0$ do
            Set $M[k,\ell] := M[k,\ell] - L[k,j] \cdot M[j,\ell]$.          /* Entry Update */
        end
    end
    Remove $j$ from $\mathcal{J}$ and $i$ from $\mathcal{I}$.      /* After this, $M[k,i] = 0$ for all $k \in \mathcal{J}$. */
end
Remove all remaining rows from $\mathcal{J}$.      /* Note these must be empty rows. */
Set $U := M$.
Return $U$, $L$, and orders in which the columns/rows were removed from $\mathcal{I}/\mathcal{J}$, respectively.
```

Algorithm 1: Gaussian elimination of $(M, \preccurlyeq)$, see Figure 1.

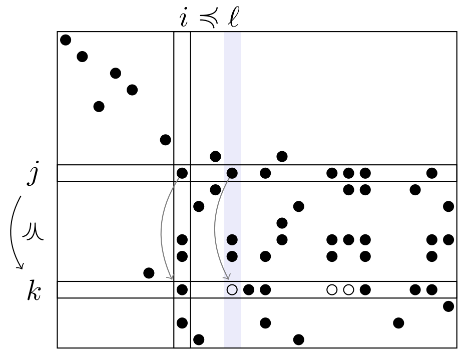


Figure 1: The positions of non-zero entries (black circles) after a few steps of the algorithm. In the current step the entry in column i, row k will be eliminated using the earlier row j – entries that will become non-zero are marked as white circles.

In Algorithm 1, we consider consecutive columns of M (in the order of $\preccurlyeq$), and for each column i we find the first row j (in the order of $\preccurlyeq$) that has a non-zero element in column i. Row j is then used to eliminate all the other non-zero entries in column i. Hence, whenever some column i is removed, it has a zero intersection with all the rows apart from the said row j. Inductively, it follows that in the Entry Update step, we have that $j \in \mathcal{J}$ with $M[j,\ell] \neq 0$ imply that $\ell \in \mathcal{I}$ and columns are never changed after removal from $\mathcal{I}$. Clearly, rows are never changed after removal.

The algorithm also returns the *removal orders* of columns and rows of M, that is, the orders in which they were removed from $\mathcal{I}$ and $\mathcal{J}$, respectively. For $\mathcal{I}$ this order coincides with the order $\preccurlyeq$, but for $\mathcal{J}$ it depends on the positioning of non-zero entries in the matrix, and can differ from $\preccurlyeq$.

We now verify that the output of the algorithm forms a suitable factorization of M and can be used for solving linear equations. In the following, we assume that L and U are given as sequences of non-zero entries in consecutive rows (in any order), so their description takes $\mathcal{O}(N)$ indices and field values, where N is the number of non-zero entries in L and U.

Lemma 4.1. *Let M be an $n \times m$ matrix over a field $\mathbb{F}$ and $\preccurlyeq$ be any ordering of its rows and columns. Within the same time bounds as needed for executing Algorithm 1, we can compute a $PLUQ$-factorization of M, and hence the rank, determinant (if $n = m$), and a maximal nonsingular submatrix of M. Furthermore, for any $r \in \mathbb{F}^n$, the system of linear equations $Mx = r$ can be solved in $\mathcal{O}(N)$ additional time and field operations (either outputting an example solution $x \in \mathbb{F}^m$ or concluding that none exists), where N is the number of non-zero entries in L and U.*

Proof. Let U, L be the matrices output by the algorithm. The sequence of row operations performed in the algorithm implies that each row $U[k, \cdot]$ of the final matrix is obtained from the row $M[k, \cdot]$ of the original matrix by adding row $U[j, \cdot]$ with multiplier $-L[k, j]$, for each j removed earlier (note that rows j removed later are never added and have $L[k, j] = 0$). That is, for each k we have $U[k, \cdot] = M[k, \cdot] - \sum_{j \neq k} L[k, j] \cdot U[j, \cdot]$, from which it immediately follows that $M = LU$ (recall that L was initialized as an identity matrix).

Let P, Q be $n \times n$ and $m \times m$ permutation matrices, respectively, such that $U' = PUQ$ is the matrix U with rows and columns reordered in the removal order output by the algorithm. Then $L' = PLP^{-1}$ is the matrix L reordered so that its rows and its columns both correspond to the rows of M in removal order. We claim that U', L'^T are in row-echelon form, and hence $M = P^{-1}L'U'Q^{-1}$ gives the desired factorization.

For matrix L', this follows directly from the fact that L has ones on the diagonal, which are preserved by the permutation P of both rows and columns, and L' is lower-triangular, because $L[k, j]$ can be non-zero only if row k is removed after row j of M in the algorithm. Hence L'^T is in row-echelon form.

For matrix U', observe that, when a row j is removed in the algorithm, it is removed together with a column i that has a non-zero intersection with j, and all rows below, that is, all rows that remain to be removed and are currently in $\mathcal{J}$, have a zero intersection with i. Hence also any column i' to the left of i has a zero intersection with row j, since j was removed strictly after removing i'. This means that $U'[j, i]$ is the first non-zero element of row j and all elements below it are zero. Hence U' is in row-echelon form.

It follows that since L' has only ones on the diagonal and since U' is in row-echelon form, the determinant of M (in case $n = m$) is equal to the product of diagonal values of U' multiplied by the signs of the permutations P and Q. Similarly, P, Q, L are of full rank, so the rank of M is the number of non-zero rows in U'. A maximum nonsingular submatrix of U' is given by the non-empty rows and the columns containing the left-most non-zero element of each such row – the same submatrix is maximal nonsingular for $L'U'$. As L' is lower-triangular with ones on the diagonal,the corresponding positions in $M = P^{-1}L'U'Q^{-1}$ give a maximal nonsingular submatrix of M. Clearly all the above can be computed within the same time bounds.

Given r, to solve $Mx = P^{-1}L'U'Q'^{-1}x = r$ it is enough to solve $Q^{-1}y = r$ trivially to get y, then $U'z = y$ to get z and similarly with L' and P^{-1} to get x. The system $U'z = y$ (for U' in row-echelon form, analogously for L') can be solved easily solved by back-substitution. That is, start from $z_i = 0$ for every column i of U'. If for any empty row j of U', y_j is non-zero, output that the system has no solutions. For every non-empty row j of U' from the lowest to the highest (so in order opposite to the removal order, with shortest rows first), let $U'[j, i]$ be the first non-zero entry of the row j and set $z_i := \frac{y_j - \sum_{i \preccurlyeq \ell} U'[j,\ell] \cdot z_\ell}{U'[j,i]}$. A standard check shows that $U'z = y$. Exactly one multiplication or division and one addition or subtraction is done for every non-zero element of U' and L'. $\square$

Using strong orderings of small width. We now introduce width parameters for the ordering $\preccurlyeq$, and show how to bound the number of fill-in entries using these parameters.

Definition 4.2. Let H be a graph. An *H-ordering* is an ordering of $V(H)$. An H-ordering $\preccurlyeq$ is *strong* if for every $i,j,k \in V(H)$ with $ij, ik \in E(H)$ and $j \preccurlyeq k$, any neighbor of j that comes after i in the ordering is also a neighbor of k; in other words, $\{\ell \in N_H(j) \mid i \preccurlyeq \ell\} \subseteq \{\ell \in N_H(k) \mid i \preccurlyeq \ell\}$. The *width* of an H-ordering $\preccurlyeq$ is defined as $\max\limits_{ij \in E(H)} \min\left(|\{\ell \in N_H(j) \mid i \preccurlyeq \ell\}|, |\{\ell \in N_H(i) \mid j \preccurlyeq \ell\}|\right)$. The *degeneracy* of an H-ordering $\preccurlyeq$ is defined as $\max\limits_{i \in V(H)} |\{\ell \in N_H(i) \mid i \preccurlyeq \ell\}|$.

Note that for any H-ordering $\preccurlyeq$, the width of $\preccurlyeq$ is upper-bounded by its degeneracy d, as for each $ij \in E(H)$, without loss of generality $i \preccurlyeq j$, we have that

$$\begin{aligned}\min\Big(|\{\ell \in N_H(j) \mid i \preccurlyeq \ell\}|, &|\{\ell \in N_H(i) \mid j \preccurlyeq \ell\}|\Big) \leq \\ &\leq |\{\ell \in N_H(i) \mid j \preccurlyeq \ell\}| \leq |\{\ell \in N_H(i) \mid i \preccurlyeq \ell\}| \leq d.\end{aligned}$$

If H is bipartite, then it can be observed that the definitions of a strong ordering and width do not depend on the relative ordering in $\preccurlyeq$ of vertices on different sides of the bipartition.

Lemma 4.3. *Let M be a matrix and let $\preccurlyeq$ be a strong H-ordering of width b, for some completion H of G_M. Consider how the matrix M is modified throughout Algorithm 1 applied on M with order $\preccurlyeq$. Then the following invariant is maintained: throughout the algorithm, H is always a completion of G_M. Furthermore, the output matrix L has at most $|E(H)| + |V(H)|$ non-zero entries.*

Proof. Zero entries of M can become non-zero only in the Entry Update step of Algorithm 1. Specifically, an entry $M[k,\ell]$ can become non-zero when $M[j,i]$, $M[k,i]$, $M[j,\ell]$ were already non-zero, $i \preccurlyeq \ell$ (by choice of i), and $j \preccurlyeq k$ (by choice of j), for certain rows and columns i,j,k,ℓ. Before this step, by inductive assumption the invariant held and thus $ij, ik \in E(H)$ and ℓ is a neighbor of j in H. Therefore, the strongness of the H-ordering $\preccurlyeq$ implies that ℓ is also a neighbor of k in H, that is $k\ell \in E(H)$. Thus, the invariant is maintained after each step.

To bound the number of non-zero entries of L, observe that every non-diagonal entry $L[k,j]$ for rows k,j of M is filled only when there is an edge $i_jk \in E(H)$, where i_j is the column together with which j was removed. Hence the number of non-zero entries in each column j of L is bounded by 1 plus the number of rows adjacent to i_j in H (or just 1 if i_j does not exist for j). Since the column i_j is different for different j, the total number of non-zero entries in L is bounded by $|E(H)| + |V(H)|$. $\square$

Implementation. We now show how to implement Algorithm 1 on a RAM machine so that the total number of field operations (arithmetic, comparison and assignment on matrix values) and time used for looping through rows and columns is small. In the following, we consider the RAM model with $\Omega(\log(n))$-bit registers (on input length n) and a second type of registers for storing field values, with an oracle that performs field operations in constant time, given positions of registers containing the field values. The input matrix M is given in any form that allows to enumerate non-zero entries (as triples indicating a row j, column i and the position of a register containing $M[j,i]$) in linear time. The ordering $\preccurlyeq$ is given in any form that allows comparison in constant time and enumeration of columns in the order in linear time (otherwise we need $\mathcal{O}(|V(H)| \log |V(H)|)$ additional time to sort the columns). The graph H is given in any form that allows to enumerate edges in linear time.

Lemma 4.4. *Let M be a matrix and let $\preccurlyeq$ be a strong H-ordering of width b, for some completion H of the bipartite graph of M. Then Algorithm 1 on M with pivoting order $\preccurlyeq$ makes $\mathcal{O}(|E(H)| \cdot b + |V(H)|)$ field operations.*

Furthermore, if H and a list of columns sorted with $\preccurlyeq$ are given, then all other operations can be done in time $\mathcal{O}(|E(H)| \cdot b + |V(H)|)$ on a RAM machine.

Proof. By enumerating all edges of H, we can construct adjacency lists for the graph in $\mathcal{O}(|E(H)|)$ time. For every vertex i of H, we store the following:

- two auxiliary values $a[i], a'[i]$, set initially to -1;
- a bit remembering whether i was removed from $\mathcal{I}$ or $\mathcal{J}$,
- and we additionally compute an array $p[i]$ containing its last b neighbors in $\preccurlyeq$ order, not necessarily sorted.

Note that all the arrays $p[i]$ can be constructed in a total of $\mathcal{O}(\sum_{i\in V(H)} |N_H[i]| \cdot b) = \mathcal{O}(|E(H)| \cdot b)$ time, by finding the last b elements of the adjacency list of i in time $\mathcal{O}(|N_H[i]| \cdot b)$, for each $i \in V(H)$.

For every edge ij of H, we store a record containing:

- the position of the registers containing values $a[i], a'[i]$, $a[j], a'[j]$ and $M[j,i]$, and
- an auxiliary value $a''[j,i]$, set initially to -1.

The record is pointed to by each occurrence of the edge in the lists and p-arrays of i and j (just as one would store an edge's weight).

Claim 4.5. *Every edge ij of H occurs in at least one of the arrays $p[i]$ or $p[j]$.*

Proof. By the definition of the width of ordering $\preccurlyeq$, either $|\{\ell \in N_H(j) \mid i \preccurlyeq \ell\}| \leq b$ or $|\{\ell \in N_H(i) \mid j \preccurlyeq \ell\}| \leq b$. In the first case, i can be found among the last b neighbors of j, symmetrically in the second case. ⌟

Note that by Claim 4.5, for each given edge ij we can access the record of ij in time $\mathcal{O}(b)$, by iterating through $p[i]$ and $p[j]$.

Let us consider the number of basic operations (field and constant-time RAM operations) executed throughout the algorithm. The outer-most loop executes one iteration for each column i in the $\preccurlyeq$ order. Since a sorted list is given, their enumeration takes $\mathcal{O}(|V(H)|)$ time. In every iteration of the outer-most loop, the rows with non-zero intersection with i are neighbors of i in the bipartite graph of M and hence in H, by Lemma 4.3. Therefore, they can be enumerated by following the adjacency list for i and comparing the corresponding entries with zero, for a total of at most $\mathcal{O}(\sum_{i\in V(H)} |N_H[i]|) = \mathcal{O}(|V(H)| + |E(H)|)$ basic operations. This also suffices for making comparisons to find the earliest row j among them.

For any fixed i, after choosing j, the inner loops perform the Entry Update step for every intersection of a row $k \in R$ and a column $\ell \in C$, where R is the set of rows in $\mathcal{J}$ having non-zero intersection with i and C is the set of columns in $\mathcal{I}$ having non-zero intersection with j. By Lemma 4.3, $R \subseteq N_H(i)$ and $C \subseteq N_H(j)$. We find R and C by iterating over $N_H(i)$ to find R, and iterating over $N_H(j)$ to find C. Note that the time spent on all these iterations amortizes to $\mathcal{O}(\sum_{i\in V(H)} |N_H(i)|) = \mathcal{O}(|E(H)|)$.

To access all the values $M[k,\ell]$ occurring at intersections $k \in R$ and $\ell \in C$, we iterate over the arrays $p[k]$ and $p[\ell]$ of each $k \in R$ and each $\ell \in C$. By Claim 4.5, each edge $k\ell$ for $k \in R, \ell \in C$ will be in at least one of these arrays (in particular $|R|\cdot|C| \leq b\cdot(|R|+|C|)$). Thus the total time used on Entry Update steps will amount to $\mathcal{O}(b \cdot (|R| + |C|))$ basic operations. More precisely, for each $k \in R$, we set $a[k] := M[k,i]$ and $a'[k] := i$. Similarly, for each $\ell \in C$, we set $a[\ell] := M[j,\ell]$ and $a'[\ell] := i$. Then, we iterate over all edges $k\ell$ with $k \in R$ and $\ell \in C$ using the p-arrays, and perform the Entry Update step for each such edge. Given k and ℓ, we can check that they are indeed in R and C by testing

$a'[k] = i$ and $a'[\ell] = i$. We can ensure that the update is not performed twice on the same entry by setting $a''[k, \ell] := i$ when the first update is performed, and then not performing it again when value i is seen in $a''[k, \ell]$. Values $M[k, i]$ and $M[j, \ell]$ can be accessed in constant time via $a[k]$ and $a[\ell]$.

Since $|R| \le |N_H(i)|$ and $|C| \le |N_H(j)|$, the total number of basic operations used for the Entry Update steps is $\mathcal{O}(b \cdot (|N_H(i)| + |N_H(j)|))$. Since i and j are afterwards removed from $\mathcal{I}$ and $\mathcal{J}$, this amortizes to $\mathcal{O}(b \cdot \sum_{i \in V(H)} |N_H(i)|) = \mathcal{O}(b \cdot |E(H)|)$ basic operations in total. Therefore, the total number of basic operations made throughout the algorithm is bounded by $\mathcal{O}(|E(H)| \cdot b + |V(H)|)$. □

4.2 Orderings for path and tree-partition decompositions

The aim of this section is to show that small path or tree-partition decompositions of the bipartite graph associated with a matrix can be used to find a completion with a strong ordering of small width. In both cases it is enough to complete the graph to a maximal graph admitting the same decomposition and take a natural ordering (corresponding to "forget nodes" – the rightmost or topmost bags that contain each vertex).

Lemma 4.6. *Given a matrix M and a path decomposition of width b of the bipartite graph $G = G_M$ of M, one can construct a completion H of G with at most $2b \cdot |V(G)|$ edges and list a strong H-ordering of degeneracy (and hence width) at most b, in time $\mathcal{O}(b \cdot |V(G)|)$.*

Proof. Consider a path decomposition of G with consecutive bags $B_1, B_2, \dots, B_q$; since we can make it a clean decomposition in linear time, we can assume that $q \le n$. For every vertex v of G, let $B_{b(v)}, B_{e(v)}$ be the first and last bag containing v, respectively. For $i = 1, 2, \dots, n$, let B'_i be the set of all the vertices v with $e(v) = i$. Let H be the graph obtained from G by adding edges between any two vertices in the same bag B_i (that is, sets B_i become cliques in H). The graph H still has pathwidth b so by Lemma 2.6 it has at most $b \cdot |V(G)|$ edges. Let $\preccurlyeq$ be any ordering that places all vertices in B'_i before all vertices B'_j, for $i < j$ (vertices within one set B'_i can be ordered arbitrarily); that is, $e(u) < e(v)$ implies $u \preccurlyeq v$. It is straightforward to perform the construction in time $\mathcal{O}(b \cdot |V(G)|)$. We claim $\preccurlyeq$ is a strong H-ordering of degeneracy b.

To show this, first observe that $uv \in E(H)$ (u and v were in a common bag of the decomposition) if and only if $b(u) \le e(v)$ and $b(v) \le e(u)$ — if $uv \in E(H)$, the vertices were in a common bag and the implication is clear, while in the other case either $b(u), e(u) < b(v)$ or $b(v), e(v) < b(u)$, giving the converse. To check strongness, let $i, j, k \in V(H)$ be such that $ij, ik \in E(H)$ and $j \preccurlyeq k$. Then $ik \in E(H)$ implies $b(k) \le e(i)$ and $j \preccurlyeq k$ implies $e(j) \le e(k)$. Let ℓ be any neighbor of j that comes after i. Then $\ell j \in E(H)$ implies $b(\ell) \le e(j)$ and $i \preccurlyeq \ell$ implies $e(i) \le e(\ell)$. Together, we have $b(\ell) \le e(j) \le e(k)$ and $b(k) \le e(i) \le e(\ell)$, hence $\ell k \in E(H)$, concluding the proof of strongness.

To bound the degeneracy of $\preccurlyeq$, for each $i \in V(H)$ we want to bound the number of neighbors ℓ of i with $i \preccurlyeq \ell$. Such a neighbor must satisfy $b(\ell) \le e(i)$ and $e(i) \le e(\ell)$. By the properties of a decomposition, ℓ must be contained in all bags from $B_{b(\ell)}$ to $B_{e(\ell)}$, hence both i and ℓ are contained in the bag $B_{e(i)}$. Therefore, the number of such neighbors ℓ is bounded by $|B_{e(i)} \setminus \{i\}| \le b$ for each $i \in V(H)$, which shows degeneracy is at most b. □

Lemma 4.7. *Given a matrix M and a tree-partition decomposition of width b of the bipartite graph G of M, one can construct a completion H of G with at most $b \cdot |V(G)|$ edges and list a strong H-ordering of degeneracy (and hence width) at most $2b$, in time $\mathcal{O}(b \cdot |V(G)|)$.*

Proof. Let $(\mathcal{T}, \{B_t\}_{t \in V(\mathcal{T})})$ be the given tree-partition decomposition of G. Let H be the graph obtained from H by adding all edges between vertices in the same or in adjacent (in $\mathcal{T}$) bags. The graph H still has tree-partition width b and hence at most $2b \cdot |V(G)|$ edges, by Corollary 2.6. For a vertex $i \in V(H)$, by $t(i)$ we denote the node of $\mathcal{T}$ whose bag contains i.

We root $\mathcal{T}$ arbitrarily, which imposes an ancestor-descendant relation on the nodes of $\mathcal{T}$. Let $\preccurlyeq$ be any ordering that goes 'upward' the decomposition, that is, places all vertices of B_t after all vertices of $B_{t'}$ for any t and its descendant t' in $\mathcal{T}$. It is straightforward to perform the construction of any such $\preccurlyeq$ in $\mathcal{O}(b \cdot |V(G)|)$. We claim $\preccurlyeq$ is a strong H-ordering of degeneracy $2b$.

The bound on degeneracy follows from the fact that the neighbors of a vertex i in H occurring later in the ordering $\preccurlyeq$ must be either in the same bag as i, or in the parent bag of the bag containing i. To show strongness, let $i, j, k \in V(H)$ be such that $ij, ik \in E(H)$ and $j \preccurlyeq k$. Let ℓ be any neighbor of j that comes after i in $\preccurlyeq$. We want to show that ℓ is a neighbor of k too. If it is not, then $t(\ell) \neq t(i)$ (as otherwise $N_H[\ell] = N_H[i] \ni k$), and similarly $t(k) \neq t(j)$ (as otherwise $N_H[k] = N_H[j] \ni \ell$). Since j and i are adjacent, $t(i)$ and $t(j)$ are either equal or adjacent (in $\mathcal{T}$).

If $t(i)$ is a child of $t(j)$, then since $t(k)$ is either equal or adjacent to $t(i)$ (due to $ik \in E(H)$) and it is not a descendant of $t(j)$ (by $j \preccurlyeq k$), it must be equal to $t(j)$, contradicting the above inequalities.

If $t(j)$ is a child of $t(i)$, then since $t(\ell)$ is either equal or adjacent to $t(j)$ (due to $j\ell \in E(H)$) and it is not a descendant of $t(i)$ (by $i \preccurlyeq \ell$), it must be equal to $t(i)$, contradicting the above inequalities.

If $t(i) = t(j)$, then $t(k)$ is either equal or adjacent to $t(i) = t(j)$ (by $ik \in E(H)$), they cannot be equal (by the above inequalities) and $t(k)$ is not a child of $t(j)$ (by $j \preccurlyeq k$), hence $t(k)$ must be the parent of $t(i) = t(j)$. Similarly, $t(\ell)$ is either equal or adjacent to $t(j) = t(i)$ (by $j\ell \in E(H)$), they cannot be equal (by the above inequalities), and $t(\ell)$ is not a child of $t(i)$ (by $i \preccurlyeq \ell$), hence $t(\ell)$ must also be the parent of $t(j) = t(i)$, implying $t(\ell) = t(k)$. Hence in any case ℓ is a neighbor of k too, proving strongness of the H-ordering $\preccurlyeq$. ☐

Given Lemmas 4.6 and 4.7, from an $n \times m$ matrix M and a path- or tree-partition- decomposition of G_M of width b, we can construct a completion H of G_M with at most $(n+m) \cdot b$ edges and a strong H-ordering of width at most $2b$. Therefore, Gaussian elimination can be performed using $\mathcal{O}((n+m) \cdot b^2)$ field operations and time by Lemma 4.4, yielding matrices with at most $(n+m) \cdot b$ non-zero entries by Lemma 4.3. Together with Lemma 4.1, this concludes the proof of Theorem 1.2.

Theorem 1.2. *Given an $n \times m$ matrix M over a field $\mathbb{F}$ and a path or tree-partition decomposition of its bipartite graph G_M of width k, Gaussian elimination on M can be performed using $\mathcal{O}(k^2 \cdot (n+m))$ field operations and time. In particular, the rank, determinant, a maximal nonsingular submatrix and a PLUQ-factorization can be computed in this time. Furthermore, for every $r \in \mathbb{F}^n$, the system of linear equations $Mx = r$ can be solved in $\mathcal{O}(k \cdot (n+m))$ additional field operations and time.*

It is tempting to try to perform the same construction as in Lemmas 4.6 and 4.7 also for standard tree decompositions that correspond to treewidth. That is, complete the graph to a chordal graph H according to the given tree decomposition, root the decomposition in an arbitrary bag, and order the vertices in a bottom-up manner according to their forget nodes (i.e., highest nodes of the tree containing them). Unfortunately, it is not hard to construct an example showing that this construction *does not* yield a strong ordering. In fact, there are chordal graph that admit no strong ordering [23]. For this reason, in the next section we show how to circumvent this difficulty by reducing the case of tree decompositions to tree-partition decompositions using the vertex splitting technique.

4.3 Vertex splitting for low treewidth matrices

In this subsection we show how vertex splitting can be used to expand a matrix of small treewidth into an equivalent matrix of small tree-partition width. This allows us to extend our results to treewidth as well and prove Theorem 1.3, without assuming the existence of a good ordering for the original matrix.

The vertex splitting operation, as described in the introduction, can be used repeatedly to change vertices into arbitrary trees, with original edges moved quite arbitrarily. While algorithms will not need to perform a series of splittings, as the final outcome in our application can be easily described and constructed directly from a given tree decomposition, we nevertheless define possible outcomes in full generality to perform inductive proofs more easily.

In this section, each tree decomposition has a tree that is arbitrarily rooted and the set of all children of each node is arbitrarily ordered, so that the following ordering can be defined. The *pre-order* of an ordered tree is the ordering of nodes which places a parent before its children and the children in the same order as defined by the tree.

A *tree-split* $\mathcal{E}$ of a graph G is an assignment of an ordered tree $\mathcal{E}(v)$ to every vertex $v \in V(G)$ and of a node pair $\mathcal{E}(uv) = (t, t')$ for every edge $uv \in E(G)$, such that $t \in V(\mathcal{E}(u))$ and $t' \in V(\mathcal{E}(v))$. A rooted tree decomposition $(\mathcal{T}, \{B_t\}_{t \in V(\mathcal{T})})$ (with an ordered tree $\mathcal{T}$) of a graph G gives rise to a tree-split $\mathcal{E}(\mathcal{T})$ as follows: for $v \in V(G)$, $\mathcal{E}(v)$ is the subtree of $\mathcal{T}$ induced by those nodes whose bags contain v; for $uv \in E(G)$, $\mathcal{E}(uv) = (t, t)$, where t is the topmost node of $\mathcal{T}$ whose bag contains both u and v (it is easy to see that there is always a unique such node). See Figure 2 for an example.

For a matrix M and a tree-split $\mathcal{E}$ of $G = G_M$, we define below the $\mathcal{E}$-split of M, denoted $M_{\mathcal{E}}$. We later show that this operation preserves the determinant up to sign, for example. We will denote the $\mathcal{E}$-split of M corresponding to a tree decomposition $\mathcal{T}$ of the bipartite graph of M simply as $M_{\mathcal{T}}$ – we aim to show that $M_{\mathcal{T}}$ preserves the algebraic properties of M and strengthens the structure given by $\mathcal{T}$ to that of a tree-partition decomposition. The ordering and sign choices in the definition are only needed to preserve the sign of the determinant.

Definition 4.8 ($\mathcal{E}$-split of M)**.** Let M be a matrix with rows $r_1, \ldots, r_n$, columns $c_1, \ldots, c_m$ and bipartite graph $G = G_M$, and let $\mathcal{E}$ be a tree-split of G. The matrix $M_{\mathcal{E}}$ has the following rows, in order: for every row r_i of the original matrix (in original order) we have a row indexed with the pair (r_i, t), where t is the root of $\mathcal{E}(r_i)$. After this, for every original row r_i we have consecutively, for every non-root node t of the tree $\mathcal{E}(r_i)$, a row indexed with the pair (r_i, t) (different t occurring according to the pre-order of the tree). Then, for every original column c_i (in original order), we have consecutively for every edge tt' of the tree $\mathcal{E}(c_i)$ a row indexed with the pair (c_i, tt') (different tt' occurring according to the pre-order of the lower node in the tree $\mathcal{E}(c_i)$). The columns of $M_{\mathcal{E}}$ are defined symmetrically, that is, they are indexed with (c_i, t) and (r_i, tt').

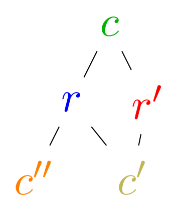
We define the entries of $M_{\mathcal{E}}$. For every non-zero entry of the original matrix, that is, every edge $r_i c_j$ of its bipartite graph, set $M_{\mathcal{E}}[(r_i, t), (c_j, t')] = M[r_i, c_j]$, where $(t, t') = \mathcal{E}(r_i c_j)$. For each row indexed as (c_i, tt'), with t being the parent of t' in $\mathcal{E}(c_i)$, set $M'[(c_i, tt'), (c_i, t')] = -M'[(c_i, tt'), (c_i, t)] = (-1)^{n'}$, where n' is the number of rows preceding (c_i, t'). Symmetrically, for each column indexed as (r_i, tt'), with t the parent of t' in $\mathcal{E}(r_i)$, set $M'[(r_i, t'), (r_i, tt')] = -M'[(r_i, t), (c_i, tt')] = \pm 1$ analogously. We set all other entries to 0. This concludes the definition.

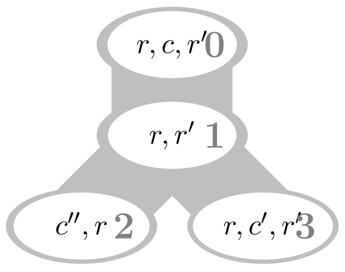
For a tree-split $\mathcal{E}$ of a graph G, we write $\|\mathcal{E}\|$ for the total number of edges in all trees $\mathcal{E}(v)$, for $v \in V(G)$. Note that if $\mathcal{E}$ is a tree-split of the bipartite graph of an $n \times m$ matrix M, then $M_{\mathcal{E}}$ has exactly $n + \|\mathcal{E}\|$ rows and $m + \|\mathcal{E}\|$ columns. For a set of rows (analogously for columns) I, we write $I_{\mathcal{E}}$ for the set of all rows of $M_{\mathcal{E}}$, except those of the form (v, t) where v is a row of M not in I and t is the root of $\mathcal{E}(v)$. In other words, $I_{\mathcal{E}}$ is obtained from I by taking rows with the same positions, and adding all the last $\|E\|$ rows of $M_{\mathcal{E}}$. In particular, $|I_{\mathcal{E}}| = |I| + \|\mathcal{E}\|$.

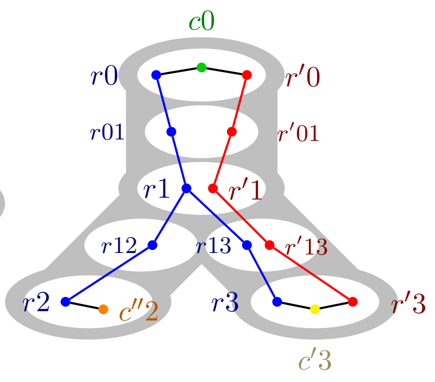
Lemma 4.9. *Let $\mathcal{E}$ be a tree-split of the bipartite graph G of an $n \times m$ matrix M. Then* $\operatorname{rk} M = \operatorname{rk} M_{\mathcal{E}} - \|\mathcal{E}\|$ *and for any sets I and J of rows and columns of M of equal size,*

$$\det[M]_{I,J} = (-1)^{\|\mathcal{E}\| \cdot (n+m)} \cdot \det[M_{\mathcal{E}}]_{I_{\mathcal{E}}, J_{\mathcal{E}}}.$$

In particular $\det M_{\mathcal{E}} = \det M$, *if* $n = m$.







	c	c'	c''
r	$\star$	$\star$	$\star$
r'	$\star$	$\star$	0

	c	c'	c''	$r01$
$r0$	$\star$	0	0	1
$r1$	0	$\star$	$\star$	-1
r'	$\star$	$\star$	0	0

	$c0$	$c'3$	$c''2$	$r01$	$r12$	$r13$	$r'01$	$r'13$
$r0$	$\star$	0	0	1	0	0	0	0
$r1$	0	0	0	-1	1	1	0	0
$r2$	0	0	$\star$	0	-1	0	0	0
$r3$	0	$\star$	0	0	0	-1	0	0
$r'0$	$\star$	0	0	0	0	0	1	0
$r'1$	0	0	0	0	0	0	-1	1
$r'3$	0	$\star$	0	0	0	0	0	-1

Figure 2: On the left: a matrix M, its bipartite graph G_M, and the matrix after splitting a row once. In the middle: a tree decomposition of G_M with nodes $0, 1, 2, 3$. On the right: the matrix after splitting according to this decomposition, and its graph in a tree-partition decomposition.

To prove the above lemma, we need the following definition and claim to perform an inductive step. For a tree-split $\mathcal{E}$ of a graph G and two adjacent nodes a, b of a tree $\mathcal{E}(v)$ assigned to some vertex $v \in V(G)$, define the *contracted* tree-split $\mathcal{E}_{v/ab}$ as $\mathcal{E}$ with the two nodes a, b identified in $\mathcal{E}(v)$ (contracting the tree edge connecting a and b) and identified in any pair $\mathcal{E}(vw)$ that contains them.

Claim 4.10. *Let $\mathcal{E}$ be a tree-split of the bipartite graph $G = G_M$ of a matrix M. Let (v, tt') be the last column or the last row of $M_\mathcal{E}$ (so t' is the last node in pre-order of the tree $\mathcal{E}(v)$ and t is its parent). Then $\operatorname{rk} M_{\mathcal{E}_{v/tt'}} = \operatorname{rk} M_\mathcal{E} - 1$ and if I, J are any sets of rows and columns of M of equal size, then $\det[M_{\mathcal{E}_{v/tt'}}]_{I_{\mathcal{E}_{v/tt'}}, J_{\mathcal{E}_{v/tt'}}} = (-1)^{n'} \cdot \det[M_\mathcal{E}]_{I_\mathcal{E}, J_\mathcal{E}}$, where n' is the number of rows (if v is a row) or columns (if v is a column) of $M_{\mathcal{E}_{v/tt'}}$.*

Proof. Without loss of generality assume v is a row of M. Note that the signs in the definition of $M_\mathcal{E}$ were chosen so that removing the last column or last row does not change any of them. Hence the matrices $M_\mathcal{E}$ and $M_{\mathcal{E}_{v/tt'}}$ only differ at rows (v, t), (v, t'), and column (v, tt'). Observe that the row (v, t) of $M_{\mathcal{E}_{v/tt'}}$ is obtained by adding rows (v, t') and (v, t) of $M_\mathcal{E}$ and deleting column (v, tt'). Thus adding row (v, t') to row (v, t) of $M_\mathcal{E}$ yields a matrix equal to $M_{\mathcal{E}_{v/tt'}}$, but with an additional column (v, tt') and an additional row (v, t'). After this row operation, the column (v, tt') has only one non-zero entry $\sigma = \pm 1$ at the intersection with row (v, t') (the other has just been canceled). Hence all other entries of this additional row can be eliminated using this entry. This makes $M_\mathcal{E}[(v, t'), (v, tt')] = \sigma$ the only non-zero entry in its row and column (after applying the above row and column operations). Hence $\operatorname{rk} M_\mathcal{E} = \operatorname{rk} M_{\mathcal{E}_{v/tt'}} + 1$.

To check minor determinants, consider any sets I, J of rows and columns of M of equal size. The sets $I_\mathcal{E}$ and $J_\mathcal{E}$ always contain (v, t') and (v, tt'), so the above row and column operations have the same effect after deleting rows and columns outside those sets, and do not change the determinant. Moving row (v,t') to the last position multiplies the determinant by $(-1)^{n_1}$, where n_1 is the number of rows below it – since $I_\mathcal{E}$ contains all rows below (v,t'), this is independent of I and J and we can count rows in $M_\mathcal{E}$ just as well. Then, deleting this last row and the last column (whose only non-zero entry is their intersection σ) multiplies the determinant by $\sigma = (-1)^{n_2}$, where n_2 was defined to count the rows originally above (v,t') in $M_\mathcal{E}$. This deletion yields a matrix equal to $M_{\mathcal{E}_{v/tt'}}$, hence $\det M_\mathcal{E} = (-1)^{n'} \cdot \det M_{\mathcal{E}_{v/tt'}}$, where $n' = n_1 + n_2$ counts all rows of $M_\mathcal{E}$ except the deleted one, which is equal to the number of all rows in $M_{\mathcal{E}_{v/tt'}}$. ⌟

Proof of Lemma 4.9. Observe that if $\mathcal{E}'$ is a trivial tree-split assigning a single node tree $\mathcal{E}(v)$ to every vertex $v \in V(G)$, then $M_{\mathcal{E}'}$ defines the same matrix as M. Hence M can be obtained from $M_\mathcal{E}$ by repeatedly contracting the edge corresponding to the last row or the last column. By Claim 4.10, the rank decreases by exactly one with every contraction, so in total it decreases by $\|\mathcal{E}\|$.

Similarly, each minor's determinant changes sign i times with every contraction, where i counts the rows or columns of the matrix after contraction. Hence in total, the number of sign changes is $\sum_{i=n+\|\mathcal{E}\|-1}^{n} i$ (for contractions corresponding to rows) plus $\sum_{i=m+\|\mathcal{E}\|-1}^{m} i$ (for columns), which is equal to $\frac{\|\mathcal{E}\|\cdot(n+\|\mathcal{E}\|-1+n)}{2} + \frac{\|\mathcal{E}\|\cdot(m+\|\mathcal{E}\|-1+m)}{2} = \|\mathcal{E}\|\cdot(n+m+\|\mathcal{E}\|-1) \equiv \|\mathcal{E}\|\cdot(n+m) \pmod 2$. □

The explicit construction of the split of a matrix allows us to easily bound its size, number of non-zero entries in each row and in total. Before this, to optimize these parameters, we need the following easy adaption of a standard lemma about constructing so called 'nice tree decompositions' (see [9]).

Lemma 4.11. *Given a tree decomposition of a graph G of width b, one can find in time $\mathcal{O}(bn)$ a tree decomposition of G of width b with a rooted tree of at most $5n$ nodes, each with at most two children. Furthermore, for every node t of the decomposition, there are at most b edges $uv \in E(G)$ such that t is topmost node whose bag contains both u and v.*

Proof. Root the given tree decomposition in an arbitrary node. For $v \in V(G)$, let $t(v)$ be the topmost node whose bag contains v – there is such a bag by properties of tree decompositions (the so called *forget* bag for v). Bodlaender, Bonsma and Lokshtanov [9, Lemma 6] describe how to transform a tree decomposition in time $\mathcal{O}(n \cdot b)$ to a 'nice tree decomposition', which in particular has the following properties: it has at most $4|V(G)|$ nodes, each with at most two children, and furthermore $t(v)$ has at most one child for each $v \in V(G)$, and for $u \neq v$, either $t(u) \neq t(v)$ or $t(u) = t(v)$ is the root of the decomposition tree. To remedy this last possibility, if the root node's bag is $B = \{b_1, \dots, b_\ell\}$, add atop of it a path of nodes with bags $\{b_1, \dots, b_i\}$ for $i = \ell, \ell-1, \dots, 1, 0$, rooted at the last, empty bag. This adds at most $\ell \leq |V(G)|$ nodes to the decomposition tree. Now $t(v) \neq t(u)$ for all vertex pairs $u \neq v \in V(G)$.

Consider now any edge $uv \in E(G)$. Since the sets of nodes whose bags contain u and those who contain v induce connected subtrees of the decomposition tree, their intersection is also a connected subtree whose topmost bag cannot be a descendant of both $t(u)$ and $t(v)$. That is, the topmost node whose bag contains both u and v is either $t(u)$ or $t(v)$. Therefore, if we assign each edge to the topmost bag that contains both its endpoints, then for each node t, the edges assigned to it must be incident to v, where $v \in V(G)$ is such that $t = t(v)$. Since such edges have both endpoints in the bag of t, there can be at most b of them. □

Lemma 4.12. *Let M be an $n \times m$ matrix with bipartite graph $G = G_M$. Let $\mathcal{T}$ be a tree decomposition of G of width b obtained from Lemma 4.11. Then $M_\mathcal{T}$ has the following properties, for some $N = \mathcal{O}(b \cdot (n+m))$:*

(a) $M_\mathcal{T}$ has $n+N$ rows and $m+N$ columns,

(b) Every row and column of $M_{\mathcal{T}}$ has at most $b+3$ non-zero entries, and $M_{\mathcal{T}}$ has at most $|E(G)|+4N$ such entries in total,

(c) The bipartite graph of $M_{\mathcal{T}}$ has a tree-partition decomposition of width b, with a tree that is the 1-subdivision of the tree of $\mathcal{T}$,

(d) $M_{\mathcal{T}}$ and the decomposition can be constructed in time $\mathcal{O}(N)$,

(e) $\operatorname{rk} M = \operatorname{rk} M_{\mathcal{T}} - N$,

(f) $\det[M]_{I,J} = (-1)^{N\cdot(n+m)} \cdot \det[M_{\mathcal{T}}]_{I',J'}$ for any sets I and J of rows and columns of M of equal size, where I' (J') is obtained from I (J) by adding the last N rows (columns) of $M_{\mathcal{T}}$ to it, (in particular $\det M_{\mathcal{T}} = \det M$, if $n = m$),

Proof. Let $\mathcal{E}$ be the tree-split corresponding to $\mathcal{T}$, let $M_{\mathcal{T}} = M_{\mathcal{E}}$ and let $N = \|\mathcal{E}\|$. Property (a) follows directly from the definition of $M_{\mathcal{E}}$. Since by definition trees $\mathcal{E}(v)$ are subtrees of $\mathcal{T}$, and, by the definition of a decomposition's width, at most $b+1$ such subtrees can share each edge of this tree, we have that $N = \|\mathcal{E}\| = \mathcal{O}(b \cdot (n+m))$.

Property (b) follows from the fact that every row of $M_{\mathcal{T}}$ is either indexed as (c, tt') (for some column c of M) — in which case it has exactly two non-zero entries — or it is indexed as (r,t) for some row r of M and some node t of $\mathcal{T}$ whose bag contains r. The only non-zero entries of row (r,t) are: $M_{\mathcal{T}}[(r,t),(c,t)]$ for edges $rc \in E(G)$ such that $\mathcal{E}(rc) = (t,t)$; and $M_{\mathcal{T}}[(r,t),(r,tt')]$ for neighbors t' of t in $\mathcal{T}$. By the guarantees of Lemma 4.11, there are at most b edges rc such that t is the topmost bag containing both endpoints, that is, such that $\mathcal{E}(rc) = (t,t)$. Furthermore, every node has at most three neighbors in the tree of $\mathcal{T}$, thus the row (r,t) has at most $b+3$ non-zero entries in total. The proof is symmetric for columns. Similarly, the total number of non-zero entries can be bounded by $2N$ for entries in rows indexed as (c,tt'), $2N$ for entries in columns indexed as (r,tt') and $|E(G)|$ for the remaining entries, which must be of the form $M_{\mathcal{T}}[(r,t),(c,t)]$ where $rc \in E(G)$ and $(t,t) = \mathcal{E}(rc)$.

Properties (c) and (d) follow easily from the construction: the tree-partition decomposition of $M_{\mathcal{T}}$'s bipartite graph will have a bag V_t for every node t of $\mathcal{T}$ and a bag $V_{tt'}$ for every edge tt' of $\mathcal{T}$ – these bags are naturally assigned to the nodes of the 1-subdivision of $\mathcal{T}$'s tree. Each bag V_t contains all rows and columns indexed as (v,t), for some $v \in V(G)$ (contained in the bag of t in $\mathcal{T}$) and each bag $V_{tt'}$ contains all rows and columns indexed as (v,tt'), for some $v \in V(G)$ (contained in both bags of t,t' in $\mathcal{T}$). This defines a valid tree-partition decomposition of width at most b, as non-zero entries are either of the form $M_{\mathcal{T}}[(r,t),(c,t)]$ and hence its row and column fall into the same bag V_t, or of the form $M_{\mathcal{T}}[(r,t),(r,tt')]$ (or symmetrically for columns) and hence its row and column fall into adjacent bags V_t, $V_{tt'}$.

Properties (e) and (f) follow directly from Lemma 4.9. □

We remark that the construction of Lemma 4.12 actually preserves the symmetry of the matrix, i.e., if M is symmetric then so is $M_{\mathcal{E}}$ as well. The construction can be also easily adapted to preserve skew-symmetry, if needed.

As a final ingredient for Theorem 1.3 we need to provide a generalized LU-factorization for the original matrix and retrieve a maximal nonsingular submatrix.

Lemma 4.13. *Let M be an $n \times m$ matrix over a field $\mathbb{F}$ and let $\mathcal{E}$ be a tree-split of the bipartite graph $G = G_M$. Given a $PLUQ$-factorization of $M_{\mathcal{E}}$ with N non-zero entries in total, a generalized LU-factorization of M with at most $N' = N + 2\|\mathcal{E}\| + 2n + 2m$ non-zero entries can be constructed in $\mathcal{O}(N')$ time. Hence given any vector $r \in \mathbb{F}^m$, the system of linear equation $Mx = r$ can be solved in $\mathcal{O}(N')$ additional field operations and time.*

Proof. Define $U_{\mathcal{E}}$ as the following matrix with n rows corresponding to rows of M and $n + \|\mathcal{E}\|$ columns corresponding to rows of $M_{\mathcal{E}}$ (indexed as (r,t) or (c,tt')). The only non-zero entries of $U_{\mathcal{E}}$ are $U_{\mathcal{E}}[r,(r,t)] = 1$ for rows r of M and nodes t of the tree $\mathcal{E}(r)$. Analogously, define $L_{\mathcal{E}}$ as the following matrix with m columns corresponding to columns of M and $m + \|\mathcal{E}\|$ rows corresponding to columns of $M_{\mathcal{E}}$ (indexed as (c,t) or (r,tt')). The only non-zero entries of $L_{\mathcal{E}}$ are $L_{\mathcal{E}}[(c,t),c] = 1$ for columns c of M and nodes t of the tree $\mathcal{E}(c)$. It is straightforward from the definition of $M_{\mathcal{E}}$ that $M = U_{\mathcal{E}} M_{\mathcal{E}} L_{\mathcal{E}}$.

Let P, L, U, Q define the given $PLUQ$-factorization of $M_{\mathcal{E}}$; then $M = U_{\mathcal{E}} PLUQL_{\mathcal{E}}$. Observe that for any ordering of columns of $U_{\mathcal{E}}$, the matrix can be given in row-echelon form by ordering rows so that r_1 comes before r_2 whenever the first column indexed as (r_1,t) (for $t \in \mathcal{E}(r_1)$) comes before the first column indexed as (r_2,t') (for $t' \in \mathcal{E}(r_2)$). Hence we can construct from P a permutation $n \times n$ matrix P' such that $U' = P'U_{\mathcal{E}}P$ is in row-echelon form. Similarly we can find an $m \times m$ permutation matrix Q' and a reordering L' of $L_{\mathcal{E}}$ such that $L' = QL_{\mathcal{E}}Q'$ is in column-echelon form. Then $M = P'^{-1}U'LUL'Q'^{-1}$ gives a generalized LU-factorization. The number of non-zero entries in $L_{\mathcal{E}}$ and in $U_{\mathcal{E}}$ is equal to $n + \|\mathcal{E}\|$ and $m + \|\mathcal{E}\|$, respectively. Hence the total number of non-zero entries in the factorization is at most $N' = 2n + \|E\| + N + \|E\| + 2m$.

To solve a system of linear equations $Mx = r$ with input vector r, we proceed with each matrix of the generalized factorization just as in Lemma 4.1, either using back-substitution or permuting entries, using a total number of field operations equal to twice the number of non-zero entries and $\mathcal{O}(N')$ time. $\square$

Given Lemma 4.12, from an $n \times m$ matrix M and a tree decomposition of G_M of width at most b, we can construct an $(n+N) \times (m+N)$ matrix $M_{\mathcal{T}}$ with a tree-partition decomposition of width at most b, in time $\mathcal{O}(N)$, for $N = \mathcal{O}((n+m) \cdot b)$. Then, by Lemma 4.7 we can find a completion H of $G_{M_{\mathcal{T}}}$ with at most $N' = (n+m+2N) \cdot b$ edges and a strong H-ordering of width at most $2b$. Therefore, Gaussian elimination can be performed on $M_{\mathcal{T}}$ using $\mathcal{O}(N' \cdot b) = \mathcal{O}((n+m) \cdot b^3)$ field operations and time by Lemma 4.4, yielding matrices with at most $2N' + n + m + 2N = \mathcal{O}((n+m) \cdot b^2)$ non-zero entries by Lemma 4.3. This allows us to retrieve the rank and determinant of $M_{\mathcal{T}}$, by Lemma 4.1, and hence also of the original matrix M, by Lemma 4.12. We can also retrieve a $PLUQ$-factorization of $M_{\mathcal{T}}$ and hence, by Lemma 4.13, a generalized LU-factorization of M with $\mathcal{O}((n+m) \cdot b^2)$ non-zero entries, which allows us to solve a system of linear equations $Mx = r$ with given r in $\mathcal{O}((n+m) \cdot b^2)$ additional field operations and time. This concludes the proof of Theorem 1.3.

Theorem 1.3. *Given an $n \times m$ matrix M over a field $\mathbb{F}$ and a tree-decomposition of its bipartite graph G_M of width k, we can calculate the rank, determinant and a generalized LU-factorization of M in $\mathcal{O}(k^3 \cdot (n+m))$ field operations and time. Furthermore, for every $r \in \mathbb{F}^n$, the system of linear equations $Mx = r$ can be solved in $\mathcal{O}(k^2 \cdot (n+m))$ additional field operations and time.*

5 Maximum matching

Computation model. As we have already mentioned in the introduction, in this section we will consider arithmetic operations in a field $\mathbb{F}$ of size $\mathcal{O}(n^c)$ for some constant c. Although any such field would suffice, let us focus our attention on $\mathbb{F} = \mathbb{F}_p$ for some prime $p = \mathcal{O}(n^c)$. Then an element of $\mathbb{F}$ can be stored in a constant number of machine words of length $\log n$, and the following arithmetic operations in $\mathbb{F}_p$ can be easily implemented in constant time, assuming constant-time arithmetic on machine words of length $\log n$: addition, subtraction, multiplication, testing versus zero. The only operation that is not easily implementable in constant time is inversion, and hence also division. In a standard RAM machine, we need to apply the extended Euclid's algorithm, which takes $\mathcal{O}(\log n)$

time. However, in real-life applications, operations such as arithmetic in a field are likely to be heavily optimized, and hence we find it useful to separate this part of the running time.

In the algorithms in the sequel we state the consumed time resources in terms of *time* (standard operations performed by the algorithm) and *field operations*, each time meaning operations in some $\mathbb{F}_p$ for a polynomially bounded p.

Computing the size of a maximum matching. To address the problem of computing the maximum matching in a graph, we need to recall some classic results on expressing this problem algebraically. Recall that a matrix A is called *skew-symmetric* if $A = -A^T$. For a graph G with vertex set $\{v_1, v_2, \ldots, v_n\}$, let $\mathcal{E} = \{x_{ij} : i < j,\, v_iv_j \in E(G)\}$ be a set of indeterminates associated with edges of G. With G we can associate the *Tutte matrix of* G, denoted $\tilde{A}(G)$ and defined as follows. The matrix $\tilde{A}(G) = [a_{ij}]_{1\le i,j\le n}$ is an $n \times n$ matrix over the field $\mathbb{Z}(\mathcal{E})$, i.e., the field of fractions of multivariate polynomials over $\mathcal{E}$ with integer coefficients. Its entries are defined as follows:

$$a_{ij} = \begin{cases} x_{ij} & \text{if } i < j \text{ and } v_iv_j \in E(G); \\ -x_{ji} & \text{if } i > j \text{ and } v_iv_j \in E(G); \\ 0 & \text{otherwise.} \end{cases}$$

Clearly $\tilde{A}(G)$ is skew-symmetric. The following result is due to Tutte [61] (for perfect matchings) and Lovász [45] (for maximum matchings).

Theorem 5.1 ([61, 45])**.** *$\tilde{A}(G)$ is nonsingular if and only if G has a perfect matching. Moreover,* $\operatorname{rk} \tilde{A}(G)$ *is equal to twice the maximum size of a matching in G.*

From Theorem 5.1 we can derive our first result on finding the cardinality of a maximum matching, that is, Theorem 1.4, which we recall below.

Theorem 1.4. *There exists an algorithm that, given a graph G together with its tree decomposition of width at most k, uses $\mathcal{O}(k^3 \cdot n)$ time and field operations and computes the size of a maximum matching in G. The algorithm is randomized with one-sided error: it is correct with probability at least $1-1/n^c$ for an arbitrarily chosen constant c, and in the case of an error it reports a suboptimal value.*

Proof. Arbitrarily enumerate $V(G)$ as $\{v_1, v_2, \ldots, v_n\}$. Let $\tilde{A}(G)$ be the Tutte matrix of G. Let p be a prime with $n^{c+1} \le p < 2 \cdot n^{c+1}$, and let $\mathbb{F} = \mathbb{F}_p$ (p can be found in polylogarithmic expected time by iteratively sampling a number and checking whether it is prime using the AKS algorithm).

Construct matrix $A(G)$ from $\tilde{A}(G)$ by substituting each indeterminate from $\mathcal{E}$ with a value chosen uniformly and independently at random from $\mathbb{F}$. Since the determinant of the largest nonsingular square submatrix of $\tilde{A}(G)$ is a polynomial over $\mathcal{E}$ of degree at most n, from Schwarz-Zippel lemma it follows that this submatrix remains nonsingular in $A(G)$ with probability at least $1 - \frac{n}{p} \ge 1 - \frac{1}{n^c}$. Moreover, for any square submatrix of $A(G)$ that is nonsingular, the corresponding submatrix of $\tilde{A}(G)$ is nonsingular as well. Hence, with probability at least $1 - \frac{1}{n^c}$ we have that matrix $A(G)$ has the same rank as $\tilde{A}(G)$, and otherwise the rank of $A(G)$ is smaller than that of $\tilde{A}(G)$.

Let H be the bipartite graph $G_{A(G)}$ associated with matrix $A(G)$. Then H has $2n$ vertices: for every vertex u of G, H contains a *row-copy* and a *column-copy* of u. Based on a decomposition of G of width k, it is easy to construct a tree decomposition of H of width at most $2k + 1$ as follows: the decomposition has the same tree, and in each bag we replace each vertex of G by both its copies in H. The construction of H and its tree decomposition takes time $\mathcal{O}(kn)$.

We now use the algorithm of Theorem 1.3 to compute the rank of $A(G)$; this uses $\mathcal{O}(k^3 \cdot n)$ time and field operations. Supposing that $A(G)$ indeed has the same rank as $\tilde{A}(G)$ (which happens with probability at least $1 - \frac{1}{n^c}$), we have by Theorem 5.1 that the size of a maximum matching in G is equal to the half of this rank, so we report this value. In the case when $\operatorname{rk} A(G) < \operatorname{rk} \tilde{A}(G)$, which happens with probability at most $\frac{1}{n^c}$, the algorithm will report a value smaller than the maximum size of a matching in G. $\square$

Reconstructing a maximum matching. Theorem 1.4 gives only the maximum size of a matching, but not the matching itself. To recover the maximum matching we need some extra work.

First, we need to perform an analogue of the splitting operation from Section 4 in order to reduce the case of tree decompositions to tree-partition decompositions. The reason is that Theorem 1.3 does not give us a maximum nonsingular submatrix of the Tutte matrix of the graph, which we need in order to reduce finding a maximum matching to finding a perfect matching in a subgraph.

Lemma 5.2. *There exists an algorithm that given a graph G together with its clean tree decomposition $(\mathcal{T}, \{B_x\}_{x \in V(\mathcal{T})})$ of width at most k, constructs another graph G' together with its tree-partition decomposition $(\mathcal{T}', \{C_x\}_{x \in V(\mathcal{T}')})$ of width at most k, such that the following holds:*

(i) The algorithm runs in time $\mathcal{O}(kn)$;

(ii) $|V(G')| \le 2kn$;

(iii) Given a matching M in G, one can construct in $\mathcal{O}(kn)$ time a matching M' in G' with $|M'| = |M| + \Lambda/2$, where $\Lambda = |V(G')| - |V(G)|$;

(iv) Given a matching M' in G', one can construct in $\mathcal{O}(kn)$ time a matching M in G with $|M| \ge |M'| - \Lambda/2$.

Proof. The vertex set of G' consists of the following vertices:

- For every vertex $u \in V(G)$ and every node $t \in V(\mathcal{T})$ with $u \in B_t$, create a vertex (u, t);
- For every vertex $u \in V(G)$ and every pair of adjacent nodes $t, t' \in V(\mathcal{T})$ with $u \in B_t \cap B_{t'}$, create a vertex (u, tt').

The edge set of G' is defined as follows:

- For every vertex $u \in V(G)$ and every pair of adjacent nodes $t, t' \in V(\mathcal{T})$ with $u \in B_t \cap B_{t'}$, add edges $(u, tt')(u, t)$ and $(u, tt')(u, t')$.
- For every edge $uv \in E(G)$ and every node $t \in V(\mathcal{T})$ with $\{u, v\} \subseteq B_t$, create an edge $(u, t)(v, t)$.

This finishes the description of G'. Construct $\mathcal{T}'$ from $\mathcal{T}$ by subdividing every edge of $\mathcal{T}$ once, i.e., $\mathcal{T}'$ is the 1-subdivision of $\mathcal{T}$. Let the subdivision node used to subdivide edge tt' be also called tt'. Place all the vertices of G' of the form (u, t) in a bag C_t, for each $t \in V(\mathcal{T})$, and place all the vertices of G' of the form (u, tt') in a bag $C_{tt'}$, for each $tt' \in E(\mathcal{T})$. Then it readily follows that $(\mathcal{T}', \{C_x\}_{x \in V(\mathcal{T}')})$ is a tree-partition decomposition of G' of width at most k. Moreover, since $\mathcal{T}$ was clean, it has at most most n nodes, and hence also at most $n-1$ edges. Hence G' has at most $2kn$ vertices, and (ii) is satisfied. It is straightforward to implement the construction above in time $\mathcal{O}(kn)$, and hence (i) also follows. We are left with showing how matchings in G and G' can be transformed one to the other.

Before we proceed, let us introduce some notation. Let W^V be the set of all the vertices of G' of the form (u, t) for some $t \in V(\mathcal{T})$, and let W^E be the set of all the vertices of G' of the form (u, tt') for some $tt' \in E(\mathcal{T})$. Then (W^V, W^E) is a partition of $V(G')$. For some $u \in V(G)$, let T_u be the subgraph of G' induced by all the vertices with u on the first coordinate, i.e., of the form (u,t) or (u,tt'). It follows that T_u is a tree. Moreover, if we denote $W^V_u = W^V \cap V(T_u)$ and $W^E_u = W^E \cap V(T_u)$, then (W^V_u, W^E_u) is a bipartition of T_u, all the vertices of W^E_u have degrees 2 in T_u and no neighbors outside T_u, and $|W^V_u| = |W^E_u| + 1$. Then, we have that

$$|V(G')| = \sum_{u \in V(G)} |W^V_u| + |W^E_u| = \sum_{u \in V(G)} (2|W^E_u| + 1) = |V(G)| + 2 \sum_{u \in V(G)} |W^E_u|,$$

so $\Lambda/2 = \sum_{u \in V(G)} |W^E_u|$.

Claim 5.3. *For each $u \in V(G)$ and each $w \in W^V_u$, there is a matching $M_{u,w}$ in T_u that matches all the vertices of T_u apart from w. Moreover, $M_{u,w}$ can be constructed in time $\mathcal{O}(|V(T_u)|)$.*

Proof. Root T_u in w. This imposes a parent-child relation on the vertices of T_u, and in particular each vertex of W^E_u has exactly one child, which of course belongs to W^V_u. Construct $M_{u,w}$ by matching each vertex of W^E_u with its only child. Then only the root w is left unmatched in T_u. ⌟

Now, the first direction is apparent.

Claim 5.4. *Condition* (iii) *holds.*

Proof. Let M be a given matching in G. Construct M' as follows: For every $uv \in M$, pick an arbitrary node t of $\mathcal{T}$ with $\{u,v\} \subseteq B_t$, and add $(u,t)(v,t)$ to M'. After this, from each subtree T_u, for $u \in V(G)$, at most one vertex is matched so far, and it must belong to W^V_u. Let $w_u \in V(T_u)$ be this matched vertex in T_u; in case no vertex of T_u is matched so far, we take w_u to be an arbitrary vertex of W^V_u. For each $u \in V(G)$, we add to M' the matching M_{u,w_u}, which has cardinality $|W^E_u|$; this concludes the construction of M'. It is clear that M' is a matching in G', and moreover

$$|M'| = |M| + \sum_{u \in V(G)} |W^E_u| = |M| + \Lambda/2.$$

It is easy to see that a straightforward construction of M' takes time $\mathcal{O}(kn)$. ⌟

We now proceed to the proof of the second direction.

Claim 5.5. *Condition* (iv) *holds.*

Proof. Let M' be a given matching in G'. We first transform it to a matching M'' in G' such that $|M''| \geq |M'|$ and for each $u \in V(G)$ at most one vertex from T_u is matched in M' with a vertex outside T_u. If for some vertex u this is not true, we arbitrarily select one edge ww' of those in the matching with exactly one endpoint, say w, in T_u (and hence in W^V_u). We then remove all edges incident to T_u from the matching except ww' and add $M_{u,w} \subseteq E(T_u)$ from Claim 5.3 to the matching. We removed at most $|W^V_u| - 1$ edges, as every edge of a matching incident to T_u must contain a vertex of W^V_u. However, we added $|W^E_u| = |W^V_u| - 1$ edges, hence the matching could only increase in size.

Construct the matching M in G as follows: Inspect all the edges of M'', and for each edge of the form $(u,t)(v,t) \in M''$ for some $u, v \in V(G)$, add the edge uv to M. The construction of M'' ensures that M is indeed a matching in G, and no edge of G is added twice to M. Moreover, for each $u \in V(G)$, we have that $M'' \cap E(T_u)$ is a matching in T_u, so in particular $|M'' \cap E(T_u)| \leq |W^E_u|$. Hence, from the construction we infer that

$$|M'| \leq |M''| = |M| + \sum_{u \in V(G)} |M'' \cap E(T_u)| \leq |M| + \sum_{u \in V(G)} |W^E_u| = |M| + \Lambda/2.$$

A straightforward implementation of the construction of M runs in time $\mathcal{O}(kn)$. ⌟

Claims 5.4 and 5.5 conclude the proof. $\square$

Next, we need to recall how finding a maximum matching can be reduced to finding a perfect matching using a theorem of Frobenius and the ability to efficiently compute a largest nonsingular submatrix; this strategy was used e.g. by Mucha and Sankowski in [49].

Theorem 5.6 (Frobenius Theorem)**.** *Suppose A is an $n \times n$ skew-symmetric matrix, and suppose $X, Y \subseteq \{1, \dots, n\}$ are such that $|X| = |Y| = \operatorname{rk} A$. Then*

$$\det[A]_{X,X} \cdot \det[A]_{Y,Y} = (-1)^{|X|} \cdot (\det[A]_{X,Y})^2 .$$

Corollary 5.7. *Let G be a graph with vertex set $\{v_1, v_2, \dots, v_n\}$, and suppose $[\tilde{A}(G)]_{X,Y}$ is a maximal nonsingular submatrix of $\tilde{A}(G)$. Then $\overline{X} = \{v_i \colon i \in X\}$ is a subset of $V(G)$ of maximum size for which $G[\overline{X}]$ contains a perfect matching.*

Proof. By the assumption about X we have that $|X| = \operatorname{rk} \tilde{A}(G)$. Since $[\tilde{A}(G)]_{X,Y}$ is nonsingular, by the Frobenius Theorem both $[\tilde{A}(G)]_{X,X}$ and $[\tilde{A}(G)]_{Y,Y}$ are nonsingular as well. Observe that $[\tilde{A}(G)]_{X,X} = \tilde{A}(G[\overline{X}])$, so by Theorem 5.1 we infer that $G[\overline{X}]$ contains a perfect matching. Moreover, $\overline{X}$ has to be the largest possible set with this property, because $|\overline{X}| = \operatorname{rk} \tilde{A}(G)$. $\square$

Now we are ready to prove the analogue of Theorem 1.5 for tree-partition width. This proof contains the main novel idea of this section. Namely, it is known if a perfect matching is present in the graph, then from the inverse of the Tutte matrix one can derive the information about which edges are contained in some perfect matching. Finding one column of the inverse amounts to solving one system of linear equations, so in time roughly $\mathcal{O}(k^2 n)$ we are able to find a suitable matching edge for any vertex of the graph. Having found such an edge, both the vertex and its match can be removed; we call this operation *ousting*. However, ousting iteratively on all the vertices would result in a quadratic dependence on n. Instead, we apply a Divide & Conquer scheme: we first oust the vertices of a balanced bag of the give tree-partition decomposition, and then recurse on the connected components of the remaining graph. This results in $\mathcal{O}(\log n)$ levels of recursion, and the total work used on each level is $\mathcal{O}(k^3 \cdot n)$. We remark that in Section 6 we apply a similar approach to the maximum flow problem.

Lemma 5.8. *There exists an algorithm that, given a graph G together with its tree-partition decomposition of width at most k, uses $\mathcal{O}(k^3 \cdot n \log n)$ time and field operations and computes a maximum matching in G. The algorithm is randomized with one-sided error: it is correct with probability at least $1 - \frac{1}{n^c}$ for an arbitrarily chosen constant c, and in the case of an error it reports a failure or a sub-optimal matching.*

Proof. Let $\{v_1, v_2, \dots, v_n\}$ be an arbitrary enumeration of the vertices of G and let $\tilde{A}(G)$ be the Tutte matrix of G. For the entire proof we fix a field $\mathbb{F} = \mathbb{F}_p$ for some prime p with $n^{c+5} \le p < 2 \cdot n^{c+5}$. Construct $A(G)$ from $\tilde{A}(G)$ by substituting each indeterminate from $\mathcal{E}$ with a value chosen uniformly and independently at random from $\mathbb{F}$. Since the determinant of the largest nonsingular submatrix of $\tilde{A}(G)$ is a polynomial over $\mathcal{E}$ of degree at most n, from the Schwarz-Zippel lemma we obtain that this submatrix remains nonsingular in $A(G)$ with probability at least $1 - \frac{n}{n^{c+5}} = 1 - \frac{1}{n^{c+4}}$; of course, in this case $A(G)$ has the same rank as $\tilde{A}(G)$.

Similarly as in the proof of Theorem 1.4, let $H = G_{A(G)}$ be the bipartite graph associated with $A(G)$. Then, a tree-partition decomposition of H of width at most $2k$ can be constructed from the given tree-partition decomposition of G of width at most k by substituting every vertex with its row-copy and column-copy in H in the corresponding bag. Therefore, we can apply the algorithm of Theorem 1.2 to $A(G)$ with this decomposition of H, and retrieve a largest nonsingular submatrix of

$A(G)$ using $\mathcal{O}(k^2 \cdot n)$ time and field operations. Suppose that this submatrix is $[A(G)]_{X,Y}$ for some $X, Y \subseteq \{1, \ldots, n\}$. Then $|X| = |Y| = \operatorname{rk} A(G)$ and of course $[\tilde{A}(G)]_{X,Y}$ is nonsingular as well. Recall that with probability at least $1 - \frac{1}{n^{c+4}}$ we have $\operatorname{rk} A(G) = \operatorname{rk} \tilde{A}(G)$, so if this holds, then $[\tilde{A}(G)]_{X,Y}$ is also the largest nonsingular submatrix of $\tilde{A}(G)$. Then the rows of $\tilde{A}(G)$ with indices of X form a base of the subspace of $\mathbb{Z}(\mathcal{E})^n$ spanned by all the rows of $\tilde{A}(G)$. By Corollary 5.7 we infer that $\overline{X} = \{v_i : i \in X\}$ is the maximum size subset of $V(G)$ for which $G[\overline{X}]$ contains a perfect matching.

We now constrain our attention to the graph $G[\overline{X}]$, and our goal is to find a perfect matching there. Observe that a tree-partition decomposition of $G[\overline{X}]$ of width at most k can be obtained by taking the input tree-partition decomposition of G, removing all the vertices of $V(G) \setminus \overline{X}$ from all the bags, and deleting bags that became empty. Hence, in this manner we effectively reduced finding a maximum matching to finding a perfect matching, and from now on we can assume w.l.o.g. that the input graph G has a perfect matching.

If G has a perfect matching, then from Theorem 5.1 we infer that $\tilde{A}(G)$ is nonsingular. Again, construct $A(G)$ from $\tilde{A}(G)$ by substituting each indeterminate from $\mathcal{E}$ with a value chosen uniformly and independently at random from $\mathbb{F}$. Since $\det \tilde{A}(G)$ is a polynomial of degree at most n over $\mathcal{E}$, from Schwarz-Zippel lemma we infer that with probability at least $1 - \frac{n}{n^{c+5}} = 1 - \frac{1}{n^{c+4}}$ matrix $A(G)$ remains nonsingular. Similarly as before, a suitable tree-partition decomposition of the bipartite graph $H = G_{A(G)}$ can be constructed from the input tree-partition decomposition of G, and hence we can apply Theorem 1.2 to $A(G)$. In particular, we can check using $\mathcal{O}(k^2 \cdot n)$ time and field operations whether $A(G)$ is indeed nonsingular, and otherwise we abort the computations and report failure.

Call an edge $e \in E(G)$ *allowed* if there is some perfect matching M in G that contains e. The following result of Rabin and Vazirani [54] shows how it can be retrieved from the inverse of $\tilde{A}(G)$ whether an edge is allowed; it is also the cornerstone of the approach of Mucha and Sankowski [49, 48].

Claim 5.9 ([54])**.** *Edge v_iv_j is allowed if and only if the entry $(\tilde{A}(G)^{-1})[j, i]$ is non-zero.*

This very easily leads to an algorithm that identifies some allowed edge incident on a vertex.

Claim 5.10. *For any vertex $u \in V(G)$, one can find an allowed edge incident on u using $\mathcal{O}(k^2 n)$ time and field operations.*

Proof. Suppose $u = v_i$ for some $i \in \{1, \ldots, n\}$. We recover the i-th column c_i of $A(G)^{-1}$ by solving the system of equations $A(G)c_i = e_i$, where e_i is the unit vector with $1_{\mathbb{F}}$ on the i-th coordinate, and zeros on the other coordinates. Using Theorem 1.2 this takes time $\mathcal{O}(k^2 n)$, including the time needed for Gaussian elimination. Now observe that c_i must have a non-zero entry on some coordinate, say the j-th, which corresponds to an edge $v_iv_j \in E(G)$. Indeed, the i-th row r_i of $A(G)$ has non-zero entries only on the coordinates that correspond to neighbors of v_i and we have that $r_i c_i = 1_{\mathbb{F}}$, so it cannot happen that all the coordinates of c_i corresponding to the neighbors of v_i have zero values.

This means that $(A(G)^{-1})[j, i] \neq 0$, which implies that $(\tilde{A}(G)^{-1})[j, i] \neq 0$. Since $v_iv_j \in E(G)$, by Claim 5.9 this means that v_iv_j is allowed. ⌟

Let us introduce the operation of *ousting* a vertex u, defined as follows: Apply the algorithm of Claim 5.10 to find an allowed edge uv incident on u, add uv to a constructed matching M, and remove both u and v from the graph. Clearly, the resulting graph G' still has a perfect matching due to uv being allowed, so we can proceed on G'. Note that we can compute a suitable tree decomposition of G' by deleting u and v from the corresponding bags of the current tree-partition decomposition, and removing bags that became empty. By Claim 5.10, any vertex can be ousted within $\mathcal{O}(k^2 \cdot n)$ time and field operations.

We can now describe the whole algorithm. Let $(\mathcal{T}, \{B_x\}_{x \in V(\mathcal{T})})$ be the given tree-partition decomposition of G, and let $q = |V(\mathcal{T})|$; as each bag can be assumed to be non-empty, we have

$q \leq n$. Define a uniform measure μ on $V(\mathcal{T})$: $\mu(x) = 1$ for each $x \in V(\mathcal{T})$. Using the algorithm of Lemma 2.10, find in time $\mathcal{O}(q)$ a node $x \in V(\mathcal{T})$ such that each connected component of $\mathcal{T} - x$ has at most $q/2$ nodes. Iteratively apply the ousting procedure to all the vertices of B_x; each application boils down to solving a system of equations using $\mathcal{O}(k^2 \cdot n)$ time and field operations, so all the ousting procedures in total use $\mathcal{O}(k^3 \cdot n)$ time and field operations. Note that in consecutive oustings we work on a graph with fewer and fewer vertices, so after each ousting we again resample the Tutte matrix of the current graph, and perform the Gaussian elimination again.

Let G' be the graph after applying all the ousting procedures. For every subtree $\mathcal{C} \in \mathtt{cc}(\mathcal{T} - x)$, let $G_{\mathcal{C}}$ be the subgraph of G' induced by the vertices of G' that are placed in the bags of $\mathcal{C}$. Then graphs $G_{\mathcal{C}}$ for $\mathcal{C} \in \mathtt{cc}(\mathcal{T} - x)$ are pairwise disjoint and non-adjacent, and their union is G'. Since G' has a perfect matching (by the properties of ousting), so does each $G_{\mathcal{C}}$. Observe that a tree-partition decomposition $\mathcal{T}_{\mathcal{C}}$ of $G_{\mathcal{C}}$ of width at most k can be obtained by inspecting $\mathcal{C}$, removing from each bag all the vertices not belonging to $G_{\mathcal{C}}$, and removing all the bags that became empty in this manner. Hence, to retrieve a perfect matching of G it suffices to apply the algorithm recursively on all the instances $(G_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}})$, and take the union of the retrieved matchings together with the edges gathered during ousting.

We first argue that the whole algorithm runs in time $\mathcal{O}(k^3 \cdot n \log n)$. Let $\mathfrak{T}$ denote the recursion tree of the algorithm — a rooted tree with nodes labelled by instances solved in recursive subcalls, where the root corresponds to the original instance $(G, \mathcal{T})$ and the children of each node correspond to subcalls invoked when solving the instance associated with this node. Note that the leaves of $\mathfrak{T}$ correspond to instances where there is only one bag. Since the number of bags is halved in each recursive subcall, and at the beginning we have $q \leq n$ bags, we immediately obtain the following:

Claim 5.11. *The height of $\mathfrak{T}$ is at most $\log_2 n$.*

We partition the work used by the algorithm among the nodes of $\mathfrak{T}$. Each node a is charged by the work needed to (i) find the balanced node x, (ii) oust the vertices of B_x, (iii) construct the subcalls $(G_{\mathcal{C}}, \mathcal{T}_{\mathcal{C}})$ for $\mathcal{C} \in \mathtt{cc}(\mathcal{T} - x)$, (iv) gather the retrieved matchings and the ousted edges into a perfect matching of the graph. From the description above it follows that if in the instance associated with a node a the graph has n_a vertices, then the total work associated with a is $\mathcal{O}(k^3 \cdot n_a)$. Since subinstances solved at each level of the recursion correspond to subgraphs of G that are pairwise disjoint, and since there are at most $\log_2 n$ levels, the total work used by the algorithm is $\mathcal{O}(k^3 \cdot n \log n)$.

We now estimate the error probability of the algorithm. Since on each level L_i all the graphs associated with the subcalls are disjoint, we infer that the total number of subcalls is $\mathcal{O}(n \log n)$. In each subcall we apply ousting at most k times, and each time we resample new elements of $\mathbb{F}$ to the Tutte matrix, which can lead to aborting the computations with probability at most $\frac{1}{n^{c+4}}$ in case the sampling led to making the matrix singular. By union bound, no such event will occur with probability at least $1 - \frac{k \cdot n \log n}{n^{c+4}} \geq 1 - \frac{1}{n^{c+1}}$. Together with the error probability of at most $\frac{1}{n^{c+4}}$ when initially reducing finding a maximum matching to finding a perfect matching, this proves that the error probability is at most $\frac{1}{n^{c+1}} + \frac{1}{n^{c+4}} \leq \frac{1}{n^c}$. □

We can now put all the pieces together and prove Theorem 1.5, which we now restate for the reader's convenience.

Theorem 1.5. *There exists an algorithm that, given a graph G together with its tree decomposition of width at most k, uses $\mathcal{O}(k^4 \cdot n \log n)$ time and field operations and computes a maximum matching in G. The algorithm is randomized with one-sided error: it is correct with probability at least $1 - 1/n^c$ for an arbitrarily chosen constant c, and in the case of an error it reports a failure or a suboptimal matching.*

Proof. Apply the algorithm of Lemma 5.2 to construct a graph G' together with a tree-partition decomposition $\mathcal{T}'$ of G' of width at most k; this takes time $\mathcal{O}(kn)$. Apply the algorithm of Lemma 5.8 to construct a maximum matching M' in G'; this uses $\mathcal{O}(k^4 \cdot n \log n)$ time and field operations, because G' has $\mathcal{O}(kn)$ vertices. Using Lemma 5.2(iv), construct in time $\mathcal{O}(kn)$ a matching M in G of size at least $|M'| - \Lambda/2$, where $\Lambda = |V(G')| - |V(G)|$. Observe now that M has to be a maximum matching in G, because otherwise, by Lemma 5.2(iii), there would be a matching in G' of size larger than $|M| + \Lambda/2 \geq |M'|$, which is a contradiction with the maximality of M'. In case the algorithm of Lemma 5.8 reported failure or returned a suboptimal matching of G', which happens with probability at most $\frac{1}{n^c}$, the same outcome is given by the constructed procedure. □

Finally, we remark that actually the number of performed inversions in the field in the algorithms of Theorems 1.4 and 1.5 is $\mathcal{O}(kn)$ and $\mathcal{O}(k^2 n)$, respectively. Hence, if inversion is assumed to cost $\mathcal{O}(\log n)$ time, while all the other operations take constant time, the running times of Theorems 1.4 and 1.5 can be stated as $\mathcal{O}(k^3 \cdot n + k \cdot n \log n)$ and $\mathcal{O}(k^4 \cdot n \log n + k^2 \cdot n \log^2 n)$, respectively.

6 Maximum flow

In this section we prove Theorem 1.6. Let us remind that our definition of (s,t)-vertex flows corresponds to a family of vertex-disjoint paths from s to t; that is, we work with directed graphs with unit capacities on vertices.

Theorem 1.6. *There exists an algorithm that given an unweighted directed graph G on n vertices, distinct terminals $s, t \in V(G)$ with $(s,t) \notin E(G)$, and a tree decomposition of G of width at most k, works in time $\mathcal{O}(k^2 \cdot n \log n)$ and computes a maximum (s,t)-vertex flow together with a minimum (s,t)-vertex cut in G.*

Proof. Assume without loss of generality that the given tree decomposition $(\mathcal{T}, \{B_x\}_{x \in V(\mathcal{T})})$ of G is clean (otherwise clean it in linear time) and let $q \leq n$ be the number of its nodes. For a graph H containing s and t, by $\kappa_H(s,t)$ we will denote the maximum size of a flow from s to t in H.

Let L be the set of all the nodes $x \in V(\mathcal{T})$ for which $\{s,t\} \subseteq B_x$; note that L can be constructed in time $\mathcal{O}(kq)$. By the properties of a tree decomposition, L is either empty or it induces a connected subtree of $\mathcal{T}$. Let $\ell = |L|$. We will design an algorithm with running time $\mathcal{O}(k^2 \cdot n \log(\ell + 2))$, which clearly suffices, due to $\ell \leq q \leq n$.

First, we need to introduce several definitions and prove some auxiliary claims.

Claim 6.1. *If $\ell = 0$, then $\kappa_G(s,t) \leq k$.*

Proof. The assumption that $\ell = 0$ means that there is no bag where s and t appear simultaneously. Since $\mathcal{T}[s]$ and $\mathcal{T}[t]$ are disjoint connected subtrees of $\mathcal{T}$, it follows that there is an edge xy of $\mathcal{T}$ whose removal disconnects $\mathcal{T}$ into subtrees $\mathcal{T}_x$ (containing x) and $\mathcal{T}_y$ (containing y) such that $\mathcal{T}[s] \subseteq \mathcal{T}_x$ and $\mathcal{T}[t] \subseteq \mathcal{T}_y$. By the properties of a tree decomposition, we have that $B_x \cap B_y$ is an (s,t)-vertex separator. Since $B_x \neq B_y$ due to $\mathcal{T}$ being clean, we have that $|B_x \cap B_y| \leq k$. Hence in particular $\kappa(s,t) \leq |B_x \cap B_y| \leq k$. ⌟

Suppose now that $\ell > 0$ and let us fix some arbitrary node x belonging to L. Let us examine a component $\mathcal{C} \in \mathsf{cc}(\mathcal{T} - x)$, which is a subtree of $\mathcal{T}$. Let $W_{\mathcal{C}}$ consist of all the vertices $u \in V(G)$ for which $\mathcal{T}[u] \subseteq \mathcal{C}$, and let $y_{\mathcal{C}}$ be the (unique) neighbor node of x contained in $\mathcal{C}$. We call subtree $\mathcal{C}$ *important* if $L \cap V(\mathcal{C}) \neq \emptyset$, i.e., $\mathcal{C}$ has at least one node that contains both s and t; since L induces a connected subtree of $\mathcal{T}$, this is equivalent to $y_{\mathcal{C}} \in L$. Otherwise $\mathcal{C}$ is called *unimportant.* A path

P from s to t is called *expensive* if $V(P) \cap (B_x \setminus \{s,t\}) \neq \emptyset$, i.e., P traverses at least one vertex of B_x other than s and t; otherwise, P is called *cheap*.

Claim 6.2. *For any $x \in L$ and any (s,t)-vertex flow $\mathcal{F}$, the number of expensive paths in $\mathcal{F}$ is at most $k-1$.*

Proof. The paths from $\mathcal{F}$ are internally vertex-disjoint and each of them traverses a vertex of $B_x \setminus \{s,t\}$. Since $|B_x \setminus \{s,t\}| \leq k-1$, the claim follows. ⌟

Claim 6.3. *For any $x \in L$ and any cheap (s,t)-path P, there exists a subtree $\mathcal{C}_P \in \mathtt{cc}(\mathcal{T}-x)$ such that $V(P) \subseteq W_{\mathcal{C}_P} \cup \{s,t\}$. Moreover, $\mathcal{C}_P$ is important.*

Proof. By the properties of a tree decomposition, we have that the vertex set of each connected component of $G - B_x$ is contained in $W_{\mathcal{C}}$ for some subtree $\mathcal{C} \in \mathtt{cc}(\mathcal{T}-x)$. Since P is cheap, we have that $V(P) \cap B_x = \{s,t\}$, and hence all the internal vertices of P have to belong to the same connected component H of $G - B_x$. Therefore, we can take $\mathcal{C}_P$ to be the connected component of $\mathcal{T} - x$ for which $V(H) \subseteq W_{\mathcal{C}_P}$. It remains to show that $\mathcal{C}_P$ is important.

For the sake of contradiction, suppose $\mathcal{C}_P$ is not important. Let $y = y_{\mathcal{C}_P}$; then $\{s,t\} \nsubseteq B_y$. Without loss of generality suppose $t \notin B_y$, as the second case is symmetric. Since $t \in B_x$, this means that no bag of $\mathcal{C}_P$ contains t. Consequently, there is no edge from any vertex of $W_{\mathcal{C}_P}$ to t in G. However, all the internal vertices of P are contained in $W_{\mathcal{C}_P}$, which is a contradiction. ⌟

We can now describe the whole algorithm. First, the algorithm computes L in time $\mathcal{O}(kq)$. If $L = \emptyset$, then by Claim 6.1 we have that $\kappa_G(s,t) \leq k$. Hence, by starting with an empty flow and applying at most k times flow augmentation (Lemma 2.1) we obtain both a maximum (s,t)-vertex flow and a minimum (s,t)-vertex cut in time $\mathcal{O}(k \cdot (n+m)) = \mathcal{O}(k^2 \cdot n)$. Therefore, suppose that $L \neq \emptyset$.

The crux of our approach is to take x to be a node that splits L in a balanced way. For this, Lemma 2.8 will be useful. Define measure μ_L on $V(\mathcal{T})$ as follows: for $x \in V(\mathcal{T})$, let $\mu_L(x) = 1$ if $x \in L$ and $\mu_L(x) = 0$ otherwise. Since $L \neq \emptyset$, μ_L is indeed a measure. Run the algorithm of Lemma 2.8 on $\mathcal{T}$ with measure μ_L; let x be the obtained balanced node. Then for each $\mathcal{C} \in \mathtt{cc}(\mathcal{T}-x)$ we have that $|L \cap V(\mathcal{C})| = \mu_L(V(\mathcal{C})) \leq \mu_L(V(\mathcal{T}))/2 = |L|/2$. In the following, the notions of expensive and cheap components refer to components of $\mathcal{T} - x$.

For each $\mathcal{C} \in \mathtt{cc}(\mathcal{T}-x)$, decide whether $\mathcal{C}$ is important by checking whether $\{s,t\} \subseteq B_{y_{\mathcal{C}}}$; this takes at most $\mathcal{O}(kq)$ time in total. Let $\mathfrak{I} \subseteq \mathtt{cc}(\mathcal{T}-x)$ be the family of all the important subtrees of $\mathcal{T} - x$. For each $\mathcal{C} \in \mathfrak{I}$, construct an instance $(G_{\mathcal{C}}, s, t, \mathcal{T}_{\mathcal{C}})$ of the maximum vertex flow problem as follows: take $G_{\mathcal{C}} = G[W_{\mathcal{C}} \cup \{s,t\}]$, construct its tree decomposition $\mathcal{T}_{\mathcal{C}}$ from $\mathcal{C}$ by removing all the vertices not contained in $W_{\mathcal{C}} \cup \{s,t\}$ from all the bags of $\mathcal{C}$, and make it clean in linear time. Clearly, $\mathcal{T}_{\mathcal{C}}$ constructed in this manner is a tree decomposition of $G_{\mathcal{C}}$ of width at most k, and has at most half as many bags containing both s and t as $\mathcal{T}$. Moreover, a straightforward implementation of the construction of all the instances $(G_{\mathcal{C}}, s, t, \mathcal{T}_{\mathcal{C}})$ for $\mathcal{C} \in \mathfrak{I}$ runs in total time $\mathcal{O}(kn)$, due to $q \leq n$.

Now, apply the algorithm recursively to all the constructed instances $(G_{\mathcal{C}}, s, t, \mathcal{T}_{\mathcal{C}})$, for all $\mathcal{C} \in \mathfrak{I}$. Each application returns a maximum (s,t)-vertex flow $\mathcal{F}_{\mathcal{C}}$ in $G_{\mathcal{C}}$. Observe that all the internal vertices of all the paths of $\mathcal{F}_{\mathcal{C}}$ are contained in $W_{\mathcal{C}}$, and all the sets $W_{\mathcal{C}}$ for $\mathcal{C} \in \mathfrak{I}$ are pairwise disjoint. Therefore, taking $\mathcal{F}^{\mathrm{apx}} = \bigcup_{\mathcal{C} \in \mathfrak{I}} \mathcal{F}_{\mathcal{C}}$ defines an (s,t)-vertex flow, because the paths of $\mathcal{F}^{\mathrm{apx}}$ are internally vertex-disjoint.

Now comes the crucial observation: $\mathcal{F}_{\mathrm{apx}}$ is not far from the maximum flow.

Claim 6.4. *$|\mathcal{F}^{apx}| \geq \kappa_G(s,t) - (k-1)$.*

Proof. Let $\mathcal{F}^\circ$ be a maximum (s,t)-vertex flow. Partition $\mathcal{F}^\circ$ as $\mathcal{F}^\circ_{\mathrm{exp}} \uplus \biguplus_{\mathcal{C}\in\mathfrak{I}} \mathcal{F}^\circ_{\mathcal{C}}$, where $\mathcal{F}^\circ_{\mathrm{exp}}$ is the set of the expensive paths from $\mathcal{F}^\circ$, whereas for $\mathcal{C} \in \mathfrak{I}$ we define $\mathcal{F}^\circ_{\mathcal{C}}$ to be the set of cheap paths from $\mathcal{F}^\circ$ whose internal vertices are contained in $W_{\mathcal{C}}$. Claim 6.3 ensures us that, indeed, each path of $\mathcal{F}^\circ$ falls into one of these sets. By Claim 6.2 we have that

$$|\mathcal{F}^\circ_{\mathrm{exp}}| \le k-1. \tag{1}$$

On the other hand, for each $\mathcal{C} \in \mathfrak{I}$ we have that

$$|\mathcal{F}_{\mathcal{C}}| = \kappa_{G_{\mathcal{C}}}(s,t) \ge |\mathcal{F}^\circ_{\mathcal{C}}|. \tag{2}$$

By combining (1) and (2), we infer that

$$|\mathcal{F}^{\mathrm{apx}}| = \sum_{\mathcal{C}\in\mathfrak{I}} |\mathcal{F}_{\mathcal{C}}| \ge \sum_{\mathcal{C}\in\mathfrak{I}} |\mathcal{F}^\circ_{\mathcal{C}}| = |\mathcal{F}^\circ| - |\mathcal{F}^\circ_{\mathrm{exp}}| \ge \kappa_G(s,t) - (k-1).$$

⌟

From Claim 6.4 it follows that in order to compute a maximum (s,t)-vertex flow and a minimum (s,t)-vertex cut in G, it suffices to start with the flow $\mathcal{F}^{\mathrm{apx}}$ and apply flow augmentation (Lemma 2.1) at most $k-1$ times. This takes time $\mathcal{O}(k^2 \cdot n)$, since by Lemma 2.6, G has at most kn edges. The algorithm clearly terminates, because the number of bags containing both s ant t is strictly smaller in each recursive subcall. From the description it is clear that it correctly reports a maximum (s,t)-vertex flow and a minimum (s,t)-vertex cut in G. We are left with estimating the running time of the algorithm.

Let $\mathfrak{T}$ denote the recursion tree of the algorithm — a rooted tree with nodes labelled by instances solved in recursive subcalls, where the root corresponds to the original instance $(G,s,t,\mathcal{T})$ and the children of each node correspond to subcalls invoked when solving the instance associated with this node. Note that the leaves of $\mathfrak{T}$ correspond to instances where there is only one bag containing both s and t. Since the number of bags containing both s and t is halved in each recursive subcall, we immediately obtain the following:

Claim 6.5. *The height of $\mathfrak{T}$ is at most $\log_2 \ell$.*

Let $d \le \log_2 \ell$ be the height of $\mathfrak{T}$, and let L_i be the set of nodes of $\mathfrak{T}$ contained on level i (at distance i from the root), for $0 \le i \le d$.

Similarly as in the proof of Theorem 1.1, we partition the work used by the algorithm among the nodes of $\mathfrak{T}$. Each node a is charged by the work needed to (i) find the balanced node x, (ii) investigate the connected components of $\mathcal{T} - x$ and find the important ones, (iii) construct the subcalls $(G_{\mathcal{C}}, s, t, \mathcal{T}_{\mathcal{C}})$ for $\mathcal{C} \in \mathfrak{I}$, (iv) construct $\mathcal{F}^{\mathrm{apx}}$ by putting together the flows returned from subcalls, and (v) run at most $k-1$ iterations of flow augmentation to obtain a maximum flow and a minimum cut. From the description above it follows that if in the instance associated with a node a the graph has n_a vertices (excluding s and t), then the total work associated with a is $\mathcal{O}(k^2 \cdot n_a)$. Note that here we use the fact that $n_a > 0$, because in every subproblem solved by the algorithm there is at least one vertex other than s and t. This follows from the fact that each leaf of a clean decomposition in any subcall contains a vertex that does not belong to the leaf's parent (or any other bag).

Obviously, each subinstance solved in the recursion corresponds to some subgraph of G. For a level i, $0 \le i \le d$, examine all the nodes $a \in L_i$. Observe that in the instances corresponding to these nodes, all the considered subgraphs share only s and t among the nodes of G. This proves that $\sum_{a\in L_i} n_a \le n$. Consequently, the total work associated with the nodes of L_i is $\sum_{a\in L_i} \mathcal{O}(k^2 \cdot n_a) \le \mathcal{O}(k^2 \cdot n)$. By Claim 6.5 the number of levels is at most $\log_2 \ell$, so the total work used by the algorithm is $\mathcal{O}(k^2 \cdot n \log \ell) \le \mathcal{O}(k^2 \cdot n \log n)$. □

As mentioned in the introduction, the single-source single-sink result of Theorem 1.6 can be easily generalized to the multiple-source multiple-sink setting. When we want to compute a maximum (S,T)-vertex flow together with a minimum (S,T)-vertex cut, it suffices to collapse the whole sets S and T into single vertices s and t, and apply the algorithm to s and t. It is easy to see that this operation increases the treewidth of the graph by at most 2, because the new vertices can be placed in every bag of the given tree decomposition. Similarly, in the setting when the minimum cut can contain vertices of $S \cup T$, which corresponds to finding the maximum number of completely vertex-disjoint paths from S to T, it suffices to add two new vertices s and t, and introduce edges (s,u) for all $u \in S$ and (v,t) for all T. Again, the new vertices can be placed in every bag of the given tree decomposition, which increases its width by at most 2.

7 Conclusions

In this work we have laid foundations for a systematic exploration of fully-polynomial FPT algorithms on graphs of low treewidth. For a number of important problems, we gave the first such algorithms with linear or quasi-linear running time dependence on the input size, which can serve as vital primitives in future works. Of particular interest is the new pivoting scheme for matrices of low treewidth, presented in Section 4, and the general Divide & Conquer approach based on pruning a balanced bag, which was used for reconstructing a maximum matching (Theorem 1.5) and computing the maximum vertex flow (Theorem 1.6).

We believe that this work is but a first step in a much larger program, since our results raise a large number of very concrete research questions. In order to facilitate further discussion, we would like to state some of them explicitly in this section.

In Section 3 we have designed an approximation algorithm for treewidth that yields an $\mathcal{O}(OPT)$-approximation in time $\mathcal{O}(k^7 \cdot n \log n)$. We did not attempt to optimize the running time dependence on k; already a better analysis of the current algorithm shows that the recursion tree in fact has depth $\mathcal{O}(k \cdot \log n)$, which gives an improved running time bound of $\mathcal{O}(k^6 \cdot n \log n)$. It is interesting whether this dependence on k can be reduced significantly, say to $\mathcal{O}(k^3)$. However, we believe that much more relevant questions concern improving the approximation factor and removing the $\log n$ factor from the running time bound.

Q1. Is there an $\mathcal{O}(OPT)$-approximation algorithm for treewidth with running time $\mathcal{O}(k^3 \cdot n \log n)$?

Q2. Is there an $\mathcal{O}(\log^c OPT)$-approximation algorithm for treewidth with running time $\mathcal{O}(k^d \cdot n \log n)$, for some constants c and d?

Q3. Is there an $\mathcal{O}(OPT^c)$-approximation algorithm for treewidth with running time $\mathcal{O}(k^d \cdot n)$, for some constants c and d?

In Section 4 we have presented a new pivoting scheme for Gaussian elimination that is based on the prior knowledge of a suitable decomposition of the row-column incidence graph. The scheme works well for decompositions corresponding to parameters pathwidth and tree-partition width, but for treewidth it breaks. We can remedy the situation by reducing the treewidth case to the tree-partition width case using ideas originating in the sparsification technique of Alon and Yuster [3], but this incurs an additional factor k to the running time. Finally, there has been a lot of work on improving the running time of Gaussian elimination using fast matrix multiplication; in particular, it can be performed in time $\mathcal{O}(n^{\omega})$ on general graphs [12] and in time $\mathcal{O}(n^{\omega/2})$ on sparse graph classes admitting with $\mathcal{O}(\sqrt{n})$ separators [3], like planar and H-minor-free graphs. When we substitute $k = n$ or $k = n^{1/2}$ in the running time of our algorithms, we fall short of these results.

Q4. Can a PLUQ-factorization of a given matrix be computed using $\mathcal{O}(k^2 n)$ arithmetic operations also when a tree decomposition of width k is given, similarly as for path and tree-partition decompositions?

Q5. Can the techniques of Hopcroft and Bunch [12] be used to obtain an $\mathcal{O}(k^c \cdot n)$-time algorithm for computing, say, the determinant of a matrix of treewidth k, so that the running time would match $\mathcal{O}(n^\omega)$ whenever $k = \Theta(n)$, and $\mathcal{O}(n^{\omega/2})$ whenever the matrix has a planar graph and $k = \Theta(n^{1/2})$?

In Section 5 we presented how our algebraic results can be used to obtain fully polynomial FPT algorithms for finding the size and constructing a maximum matching in a graph of low treewidth, where the running time dependence on the size of the graph is almost linear. In both cases, we needed to perform computations in a finite field of size poly(n), which resulted in an unexpected technicality: the appearance of an additional $\log n$ factor, depending on the computation model. We believe that this additional factor should not be necessary. Also, when reconstructing the matching itself, we used an additional $\mathcal{O}(k \log n)$ factor. Perhaps more importantly, we do not see how the presented technique can be extended to the weighted setting, even in the very simple case of only having weights 1 and 2.

Q6. Can one find the size of a maximum matching in a graph without the additional $\log n$ factor incurred by algebraic operations in a finite field of size poly(n)?

Q7. Can one construct a maximum matching in a graph of low treewidth in the same time as for finding its size?

Q8. Can one compute a maximum matching in a weighted graph in time $\mathcal{O}(k^c \cdot n \log n)$, where k is the width of a given tree decomposition and c is some constant, at least for integer weights?

In Section 6 we showed how a Divide & Conquer approach can be used to design algorithms for finding maximum vertex flows in low treewidth graphs with quasi-linear running time dependence on the size of the graph. Our technique is crucially based on the fact that we work with unweighted vertex flows, because we use the property that the size of a bag upper bounds the number of paths that can use any of its vertices. Hence, we do not see how our techniques can be extended to the setting with capacities on vertices, or to edge-disjoint flows. Of course, there is also a question of removing the $\log n$ factor from the running time bound.

Q9. Is there an algorithm for computing a maximum (s,t)-vertex flow in a directed graph with a tree decomposition of width k that would run in time $\mathcal{O}(k^c \cdot n)$ for some constant c?

Q10. Can one compute a maximum (s,t)-vertex flow in a (directed) graph with capacities on vertices in time $\mathcal{O}(k^c \cdot n)$, where k is the width of a given tree decomposition and c is some constant?

Q11. Can one compute a maximum (s,t)-edge flow in a (directed) graph in time $\mathcal{O}(k^c \cdot n \log n)$, where k is the width of a given tree decomposition and c is some constant? What about edge capacities?

Of course, one can look for other cases when developing fully polynomial FPT algorithms can lead to an improvement over the fastest known general-usage algorithms when the given instance has low treewidth. Let us propose one important example of such a question.

Q12. Can one design an algorithm for Linear Programming that would have running time $\mathcal{O}(k^c \cdot (n+m) \log(n+m))$ for some constant c, when the $n \times m$ matrix of the given program has treewidth at most k?

Finally, we remark that Giannopoulou et al. [32] proposed the following complexity formalism for fully polynomial FPT algorithms. For a polynomial function $p(n)$, we say that a parameterized problem Π is in P-FPT($p(n)$) (for *polynomial-FPT*) if it can be solved in time $\mathcal{O}(k^c \cdot p(n))$, where k is the parameter and c is some constant. The class P-FPT(n) for $p(n) = n$ is called PL-FPT (for *polynomial-linear FPT*). We can also define class PQL-FPT as $\bigcup_{d \geq 1}$ P-FPT($n \log^d n$), that is, PQL-FPT comprises problems solvable in fully polynomial FPT time where the dependence on the input size is quasi-linear. In this work we were not interested in studying any deeper complexity theory related to fully polynomial FPT algorithms, but our algorithms can be, of course, interpreted as a fundamental toolbox of positive results for PL-FPT and PQL-FPT algorithms for the treewidth parameterization. On the other hand, the results of Abboud et al. [1] on Radius and Diameter are the first attempts of building a lower bound methodology for these complexity classes. Therefore, further investigation of the complexity theory related to P-FPT classes looks like a very promising direction.

Bibliography

[1] A. Abboud, V. V. Williams, and J. R. Wang. Approximation and fixed parameter subquadratic algorithms for radius and diameter in sparse graphs. In *SODA 2016*, pages 377–391. SIAM, 2016.

[2] T. Akiba, C. Sommer, and K. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT 2012*, pages 144–155. ACM, 2012.

[3] N. Alon and R. Yuster. Matrix sparsification and nested dissection over arbitrary fields. *J. ACM*, 60(4):25, 2013.

[4] E. Amir. Approximation algorithms for treewidth. *Algorithmica*, 56(4):448–479, 2010.

[5] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[6] M. Bakonyi and A. Bono. Several results on chordal bipartite graphs. *Czechoslovak Mathematical Journal*, 47(4):577–583, 1997.

[7] N. Blum. A new approach to maximum matching in general graphs. In *ICALP 1990*, volume 443 of *Lecture Notes in Computer Science*, pages 586–597. Springer, 1990.

[8] H. L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[9] H. L. Bodlaender, P. S. Bonsma, and D. Lokshtanov. The fine details of fast dynamic programming over tree decompositions. In *IPEC 2013*, volume 8246 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 2013.

[10] H. L. Bodlaender, P. G. Drange, M. S. Dregi, F. V. Fomin, D. Lokshtanov, and M. Pilipczuk. A c^{k} n 5-approximation algorithm for treewidth. *SIAM J. Comput.*, 45(2):317–378, 2016.

[11] H. L. Bodlaender, J. R. Gilbert, H. Hafsteinsson, and T. Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms*, 18(2):238–255, 1995.

[12] J. R. Bunch and J. E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

[13] E. W. Chambers and D. Eppstein. Flows in one-crossing-minor-free graphs. *J. Graph Algorithms Appl.*, 17(3):201–220, 2013.

[14] K. Chatterjee and J. Łącki. Faster algorithms for Markov decision processes with low treewidth. In *CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 543–558. Springer, 2013.

[15] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewdith. Part II: optimal parallel algorithms. *Theor. Comput. Sci.*, 203(2):205–223, 1998.

[16] S. Chaudhuri and C. D. Zaroliagis. Shortest paths in digraphs of small treewidth. Part I: sequential algorithms. *Algorithmica*, 27(3):212–226, 2000.

[17] H. Y. Cheung, L. C. Lau, and K. M. Leung. Graph connectivities, network coding, and expander graphs. *SIAM J. Comput.*, 42(3):733–751, 2013.

[18] P. Christiano, J. A. Kelner, A. Mądry, D. A. Spielman, and S. Teng. Electrical flows, laplacian systems,

and faster approximation of maximum flow in undirected graphs. In *STOC 2011*, pages 273–282. ACM, 2011.

[19] M. B. Cohen, A. Mądry, P. Sankowski, and A. Vladu. Negative-weight shortest paths and unit capacity minimum cost flow in $\tilde{\mathcal{O}}(m^{10/7} \log w)$ time. In *SODA 2017*. SIAM, 2016. to appear, arXiv:1605.01717.

[20] B. Courcelle. The Monadic Second-Order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.

[21] M. Cygan, F. V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms*. Springer, 2015.

[22] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Doklady*, 11:1277–1280, 1970.

[23] F. F. Dragan. Strongly orderable graphs a common generalization of strongly chordal and chordal bipartite graphs. *Discrete Applied Mathematics*, 99(1-3):427–442, 2000.

[24] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[25] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[26] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

[27] U. Feige, M. Hajiaghayi, and J. R. Lee. Improved approximation algorithms for minimum weight vertex separators. *SIAM J. Comput.*, 38(2):629–657, 2008.

[28] U. Feige and M. Mahdian. Finding small balanced separators. In *STOC 2006*, pages 375–384, 2006.

[29] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, 2006.

[30] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[31] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph-matching problems. *J. ACM*, 38(4):815–853, 1991.

[32] A. C. Giannopoulou, G. B. Mertzios, and R. Niedermeier. Polynomial fixed-parameter algorithms: A case study for longest path on interval graphs. In *IPEC 2015*, volume 43 of *LIPIcs*, pages 102–113. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[33] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *J. ACM*, 45(5):783–797, 1998.

[34] M. C. Golumbic and C. F. Goss. Perfect elimination and chordal bipartite graphs. *Journal of Graph Theory*, 2(2):155–163, 1978.

[35] J. Gustedt, O. A. Mæhle, and J. A. Telle. The treewidth of java programs. In *ALENEX 2002*, volume 2409 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 2002.

[36] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde. Characterizing multiterminal flow networks and computing flows in networks of small treewidth. *J. Comput. Syst. Sci.*, 57(3):366–375, 1998.

[37] N. J. A. Harvey. Algebraic algorithms for matching and matroid problems. *SIAM J. Comput.*, 39(2):679–702, 2009.

[38] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *J. Comput. Syst. Sci.*, 63(4):512–530, 2001.

[39] A. V. Karzanov. O nakhozhdenii maksimal'nogo potoka v setyakh spetsial'nogo vida i nekotorykh prilozheniyakh. *Matematicheskie Voprosy Upravleniya Proizvodstvom*, 5:81–94, 1973. In Russian.

[40] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *SODA 2014*, pages 217–226. SIAM, 2014.

[41] J. M. Kleinberg and É. Tardos. *Algorithm design*. Addison-Wesley, 2006.

[42] Y. T. Lee, S. Rao, and N. Srivastava. A new approach to computing maximum flows using electrical flows. In *STOC 2013*, pages 755–764. ACM, 2013.

[43] F. T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *J. ACM*, 46(6):787–832, 1999.

[44] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, 1979.

[45] L. Lovász. On determinants, matchings, and random algorithms. In *Fundamentals of Computation Theory*, pages 565–574. Akademie Verlag, 1979.

[46] A. Mądry. Navigating central path with electrical flows: From flows to matchings, and back. In *FOCS 2013*, pages 253–262. IEEE Computer Society, 2013.

[47] A. Mądry. Computing maximum flow with augmenting electrical flows. In *FOCS 2016*. IEEE Computer Society, 2016. to appear, arXiv:1608.06016.

[48] M. Mucha and P. Sankowski. Maximum matchings via Gaussian elimination. In *FOCS 2004*, pages 248–255. IEEE Computer Society, 2004.

[49] M. Mucha and P. Sankowski. Maximum matchings in planar graphs via gaussian elimination. *Algorithmica*, 45(1):3–20, 2006.

[50] J. B. Orlin. Max flows in $O(nm)$ time, or better. In *STOC 2013*, pages 765–774. ACM, 2013.

[51] S. Parter. The use of linear graphs in gauss elimination. *SIAM Review*, 3(2):119–130, 1961.

[52] R. Peng. Approximate undirected maximum flows in $O(m\text{polylog}(n))$ time. In *SODA 2016*, pages 1862–1867. SIAM, 2016.

[53] L. Planken, M. de Weerdt, and R. van der Krogt. Computing all-pairs shortest paths by leveraging low treewidth. *J. Artif. Intell. Res. (JAIR)*, 43:353–388, 2012.

[54] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. *J. Algorithms*, 10(4):557–567, 1989.

[55] V. Radhakrishnan, H. B. Hunt III, and R. E. Stearns. Efficient algorithms for solving systems of linear equations and path problems. In *STACS 1992*, pages 109–119, 1992.

[56] B. A. Reed. Finding approximate separators and computing tree width quickly. In *STOC 1992*, pages 221–228. ACM, 1992.

[57] N. Robertson and P. D. Seymour. Graph Minors XIII. The Disjoint Paths problem. *J. Comb. Theory, Ser. B*, 63(1):65–110, 1995.

[58] D. J. Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597 – 609, 1970.

[59] J. Sherman. Nearly maximum flows in nearly linear time. In *FOCS 2013*, pages 263–269. IEEE Computer Society, 2013.

[60] M. Thorup. All structured programs have small tree-width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998.

[61] W. T. Tutte. The factorization of linear graphs. *J. London Math. Soc.*, 22:107–111, 1947.

[62] V. V. Vazirani. A proof of the MV matching algorithm, 2014.

[63] D. P. Williamson and D. B. Shmoys. *The design of approximation algorithms*. Cambridge university press, 2011.

[64] D. B. Wilson. Determinant algorithms for random planar structures. In *SODA 1997*, pages 258–267. ACM/SIAM, 1997.

[65] D. R. Wood. On tree-partition-width. *Eur. J. Comb.*, 30(5):1245–1253, 2009.

[66] Y. Wu, P. Austrin, T. Pitassi, and D. Liu. Inapproximability of treewidth and related problems. *J. Artif. Intell. Res. (JAIR)*, 49:569–600, 2014.

[67] R. Yuster and U. Zwick. Maximum matching in graphs with an excluded minor. In *SODA 2007*, pages 108–117. SIAM, 2007.